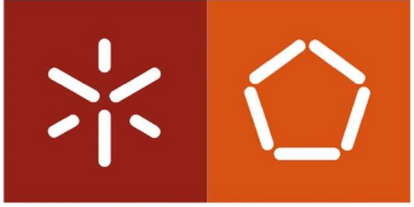
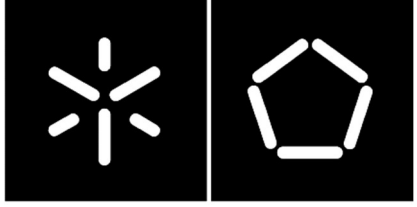


Universidade do Minho
Escola de Engenharia

Rúben Alberto Pimenta Jácome de Sousa

Desenvolvimento de um Sistema de Internet das Coisas para Comunicação com Carros Elétricos

Novembro de 2017



Universidade do Minho
Escola de Engenharia

Rúben Alberto Pimenta Jácome de Sousa

Desenvolvimento de um Sistema de Internet das Coisas para Comunicação com Carros Elétricos

Dissertação de Mestrado
Mestrado Integrado em Engenharia de Telecomunicações e Informática

Trabalho realizado sob orientação de
Professor Doutor José Augusto Afonso
Professor Doutor João Luiz Afonso

Novembro de 2017

DECLARAÇÃO

Nome: Rúben Alberto Pimenta Jácome de Sousa

Endereço eletrónico: rubensousa.mieti@gmail.com

Número do Bilhete de Identidade: 14606910

Título da dissertação: Desenvolvimento de uma Arquitetura de Internet das Coisas para Comunicação com Carros Elétricos

Orientador: Professor Doutor José Augusto Afonso

Coorientador: Professor Doutor João Luiz Afonso

Ano de conclusão: 2017

Mestrado Integrado em Engenharia de Telecomunicações e Informática

Escola: Escola de Engenharia da Universidade do Minho

Universidade do Minho, ___/___/_____

Assinatura: _____

Agradecimentos

Gostaria de agradecer ao Professor Doutor José Augusto Afonso por todo o acompanhamento ao longo do trabalho e pela sua disponibilidade para resolver todas as dúvidas que tive.

Gostaria de agradecer a todos os professores que fizeram parte do meu percurso universitário, em especial à Professora Doutora Maria João Nicolau e ao Professor Doutor Vítor Francisco Fonte, os quais me marcaram mais.

Por último, gostaria de agradecer aos meus pais, aos meus avós, à minha irmã e aos meus amigos por todo o apoio, carinho e conselhos durante o meu percurso escolar. Esta dissertação não seria concretizável sem a ajuda de todos eles.

Resumo

Com a vulgarização do acesso móvel à Internet e a diminuição do seu custo, a utilização de dados móveis tornou-se um hábito comum no dia a dia de uma pessoa, o que abre um enorme leque de possibilidades para a concretização de projetos ligados à área da Internet das Coisas. Esta dissertação visa permitir aos condutores consultarem ou controlarem um leque de parâmetros do seu carro sem necessidade de estarem perto do mesmo. Um dos parâmetros que pode ser consultado remotamente é o de nível de carga da bateria, o que permite ao condutor adequar o seu comportamento e não se esquecer de carregar o carro elétrico.

Foi desenvolvida uma arquitetura de IoT que usa uma *cloud* para sincronizar os dados do veículo em tempo real e valores de corrente de um sistema de controlo de carregamento da bateria do veículo elétrico. Para a visualização dos dados do veículo, foi desenvolvida uma aplicação móvel em Android. A aplicação recolhe dados a partir de estações periféricas que foram configuradas com um *kit* Bluetooth Low Energy (BLE) e que se ligam a sistemas sensores já instalados no veículo. Para o sistema de controlo de carregamento da bateria, foi desenvolvido um programa para o Raspberry Pi que grava o valor de corrente disponível para carregamento na *cloud* em tempo real. Foi desenvolvido também um sistema de notificações em tempo real que alerta para certos eventos como o baixo nível ou carga completa da bateria do veículo.

Abstract

The steadily increase of Internet mobile access throughout the world and its diminishing cost led to an increase of mobile data usage in people's daily lives. This opens up an enourmous range of possibilities and project ideas related to the Internet of Things.

This dissertation aims to provide a solution for remote monitoring and control of a vehicle's data and its current state without the need for the user to be inside it. One of the attributes that can be checked remotely is the battery charge, which allows the drivers to adapt their behavior and remind them to charge their electric vehicles.

An IoT architecture was developed using a cloud to synchronize the vehicle's data in real time and the current values of a battery charging control system. An Android app was developed to visualize the vehicle's data. This app connects to peripheral stations via BLE and these stations connect to sensors installed inside the vehicle. For the battery charging control system, a program for a Raspberry Pi was developed that synchronizes the available charging current for the vehicle's battery in real time to the cloud. A real time notification system was also devised to alert drivers about certain events such as low battery and full battery charge.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

BaaS Backend as a Service. 43

BLE Bluetooth Low Energy. iii, 19

CEPIUM Carro Elétrico Plug-In da Universidade do Minho. xi, 18, 116

FCM Firebase Cloud Messaging. 45

IDE Integrated Development Environment. 21, 65

IoT Internet of Things. iii, xi, 17, 21, 31, 39, 44, 48

JSON JavaScript Object Notation. 44

MQTT Message Queue Telemetry Transport. 38

MVP Model-View-Presenter. xi, 29, 30, 85, 86

OBD-II On-board Diagnostic System. 49

PNS Platform Notification Service. 37

POJOs Plain Old Java Objects. 85

RU Request Units. 35

SDK Software Development Kit. 38

UART Universal Asynchronous Receiver/Transmitter. xii, 66, 71, 72, 76

UTC Coordinated Universal Time. 57

UUID Universally Unique Identifier. 68, 69, 96

Capítulo 1

Introdução

A facilidade de acesso à Internet continua a aumentar todos os anos, com o aumento de pontos de acesso Wi-Fi e aumento de capacidade das redes celulares móveis, permitindo ligação em praticamente todos os locais. Com a vulgarização do acesso móvel à Internet e a diminuição do seu custo, a utilização de dados móveis tornou-se um hábito comum no dia a dia de uma pessoa, o que abre um enorme leque de possibilidades para a concretização de projetos ligados à área da IoT. Cerca de 47% da população mundial já usa a Internet [1] e prevê-se que em 2020 o número de dispositivos ligados à Internet será superior a 50 mil milhões [2]. Se a população mundial aumentar para os 8 mil milhões nesse mesmo ano, isto significa que teremos mais de seis dispositivos ligados à Internet por cada pessoa.

Atualmente um utilizador pode consultar dados de fontes como uma pulseira ou *smartwatch* e estar a par das suas horas de sono ou quantos quilómetros caminhou. A Internet das Coisas abriu portas para projetos que até recentemente não passavam de conteúdo de filmes de ficção científica. Hoje começamos a construir casas, cidades e veículos inteligentes que se controlam sem qualquer interação humana.

1.1 Enquadramento

Nesta dissertação foi desenvolvida um sistema de IoT aplicada a veículos elétricos. O sistema tem como base o trabalho desenvolvido em [3] e em [4] para o Carro Elétrico Plug-In da Universidade do Minho (CEPIUM), que pode ser visto na Figura 1.1.

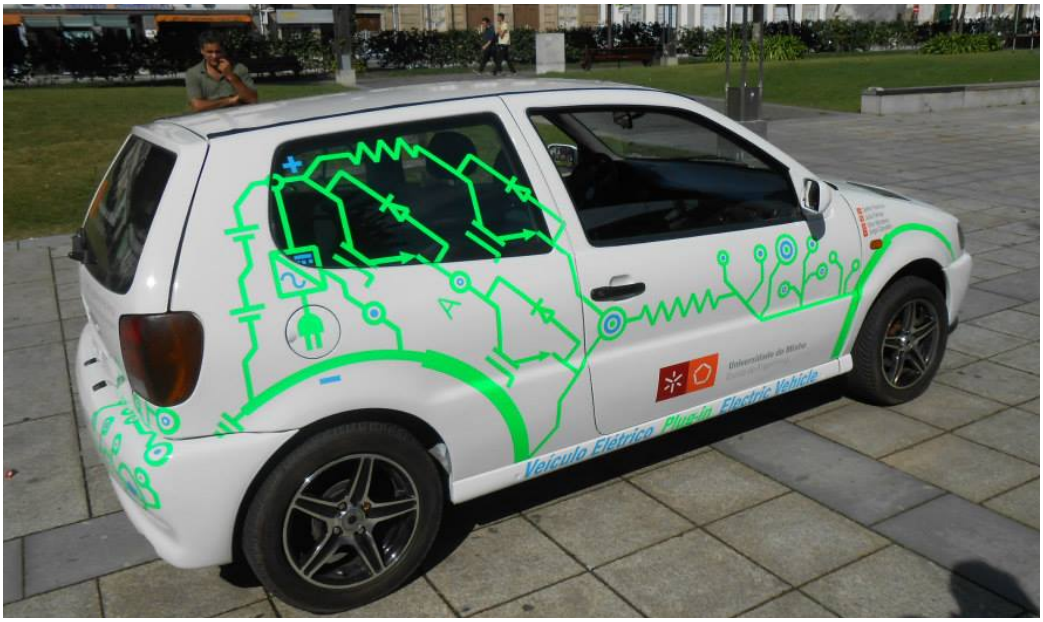


Figura 1.1: Carro Elétrico Plug-In da Universidade do Minho (CEPIUM) [5].

Atualmente, existem um conjunto de sensores implementados no próprio veículo que comunicam com uma estação base representada por um *smartphone*. No sistema implementado em [3], os dados são recebidos via BLE [6] e mostrados ao utilizador atráves de uma aplicação Android. No entanto, não existe sincronização dos dados com um servidor na Internet. Assim, o utilizador apenas consegue consultar os dados enquanto está dentro do próprio veículo. Esta dissertação vem assim complementar o trabalho desenvolvido anteriormente, trazendo melhorias na experiência do utilizador, podendo este

consultar o estado do veículo em tempo real.

1.2 Objetivos

O objetivo global desta dissertação é desenvolver uma arquitetura da Internet das Coisas usando como base o trabalho já realizado numa dissertação anterior [3]. É necessário interligar a aplicação cliente que existe atualmente a um servidor, para que os dados recolhidos estejam disponíveis em qualquer outro dispositivo, via Internet. Outro objetivo é o desenvolvimento de uma solução de carregamento inteligente, com recurso a um sensor de corrente numa habitação, ligado a um Raspberry Pi, que por sua vez estará ligado à Internet.

De forma sucinta, os objetivos são:

1. Desenvolver um servidor para sincronizar os dados da aplicação cliente.
2. Criar uma única aplicação cliente que sirva de *gateway* e que permita consultar os dados do servidor. Esta aplicação deve permitir comutar o estado de carregamento do carro via BLE.
3. Desenvolver um programa para o Raspberry Pi que leia dados do sensor de corrente da habitação e sincronize os mesmos para um servidor.
4. Elaboração de testes de todo o sistema.

O sistema desenvolvido deve ter em conta o consumo de dados móveis. O volume de dados a ser transmitido tem de ser o mínimo possível para os custos serem reduzidos e a transmissão mais eficiente. Tem também de ter em conta possíveis interrupções da ligação à Internet e garantir que a informação não se perca só porque não a foi possível transmitir.

1.3 Estrutura da Dissertação

Esta dissertação está organizada em cinco capítulos: Introdução, Estado da Arte, Desenvolvimento do Sistema, Testes, Resultados e Conclusão.

No segundo capítulo, "Estado da Arte", são apresentados conceitos e tecnologias relevantes para o desenvolvimento de todo o sistema. Em particular, são apresentados serviços de *cloud* e comparações entre os mesmos. No fim, são mencionados trabalhos relacionados.

No terceiro capítulo, "Desenvolvimento do Sistema", é descrita a arquitetura do sistema. As duas primeiras secções apresentam uma série de passos para a configuração da *cloud* Firebase e das estações periféricas BLE. Nas duas últimas secções é descrita a implementação do programa para o Rasberry Pi, que envia a corrente de carregamento para o Firebase, e da aplicação cliente, que sincroniza com o veículo.

No quarto capítulo, "Testes e Resultados", são apresentadas medições de parâmetros relevantes das aplicações, tais como o consumo de dados e atrasos na comunicação. São também apresentadas imagens da aplicação que mostram os dados recolhidos a partir de sensores emulados.

No quinto e último capítulo é feita uma discussão de todo o trabalho desenvolvido e são apresentadas sugestões de trabalho futuro.

Capítulo 2

Estado da Arte

Neste capítulo serão mencionados conceitos relevantes para o contexto dissertação, tecnologias e serviços de *cloud* disponíveis para uma arquitetura IoT. Na primeira secção são abordados conceitos relativos ao GATT. Na segunda secção são descritos os componentes de uma aplicação Android e a sua arquitetura. Nas secções seguintes são apresentados diversos serviços de *cloud* e comparações a nível de preços e caraterísticas. Por último, são apresentados projetos que involvem uma arquitetura IoT semelhante à desta dissertação.

2.1 Bluetooth GATT

O GATT (Generic Attribute Profile) do BLE define a estrutura de dados que um dado serviço de dados expõe. Este serviço é implementado num servidor GATT (GATT Server), que por sua vez comunica com um cliente ou vários (GATT Client), como pode ser visto na Figura 2.1.

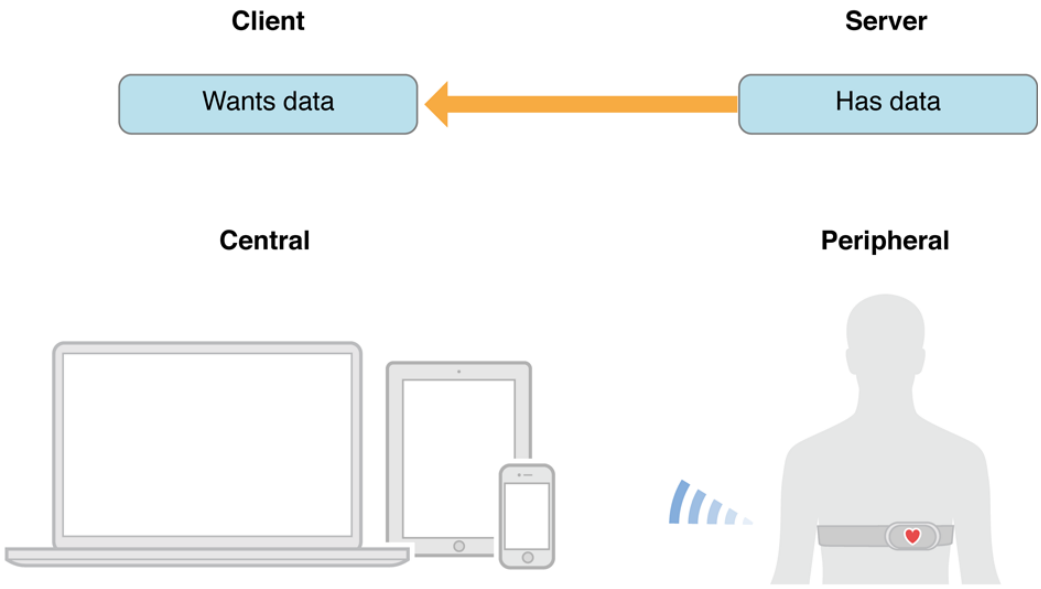


Figura 2.1: Dispositivos periféricos e centrais [7].

Os dispositivos periféricos são tipicamente associados a sensores e implementam um servidor GATT que expõe os dados medidos. Estes dispositivos ligam-se a clientes que podem ser aplicações móveis ou computadores.

A descoberta dos dispositivos periféricos é feita nos clientes através da deteção dos pacotes de anúncio enviados. Um pacote de anúncio contém informações que identificam o dispositivo periférico, tais como o nome e serviços que implementa. Por exemplo, o dispositivo periférico da Figura 2.1 anuncia que fornece o batimento cardíaco.

Cada serviço GATT é composto por um conjunto de caraterísticas, que definem os dados que podem ser lidos ou escritos, como pode ser visto na Figura 2.2.

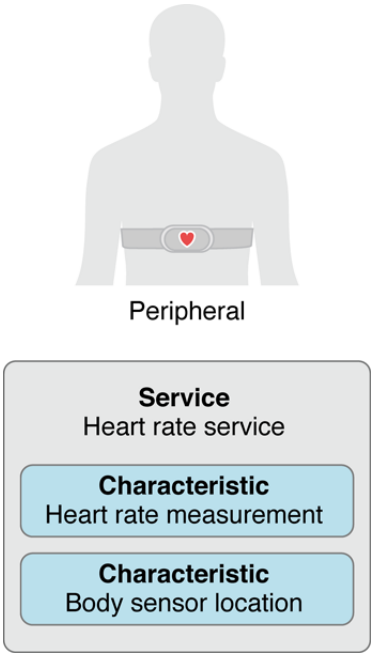


Figura 2.2: Caraterísticas de um serviço GATT [7].

Cada caraterística tem um identificador único ou Universally Unique Identifier (UUID), um campo que representa o valor, um campo para subscrição de notificações ou indicações e diversos campos para permissões de leitura e escrita. O valor da caraterística pode ser de diversos tipos (booleano, byte ou número flutuante).

Quando o dispositivo central estabelece uma ligação a um dispositivo periférico, passa a saber os serviços e as caraterísticas associadas a cada um. Depois de conhecidas as caraterísticas, o dispositivo central pode interagir com o periférico e escrever ou pedir dados através de *polling* ou notificações.

2.2 Sistema operativo Android

O Android é um sistema operativo para dispositivos móveis baseado em Linux e atualmente desenvolvido pela Google, depois de ter sido adquirido à Android Inc. em 2005. De acordo com [8], cerca de 85% de todos os *smartphones* no planeta usam o sistema operativo Android. Está também disponível para *tablets*, televisões e carros (Android Auto). A última versão, à data de escrita da dissertação, era o Android Oreo, versão 8.0. O IDE oficial usado para o desenvolvimento de aplicações nativas chama-se Android Studio [9]. Nesta secção serão mencionados os diferentes componentes que fazem parte de uma aplicação e como interagem entre si.

2.2.1 Activity

Uma Activity representa um ecrã da aplicação. Pode ser composta por várias Views e/ou Fragments que contêm outras Views. Normalmente, o ecrã principal de uma aplicação é criado com uma Activity chamada MainActivity. Todas as Activities de uma aplicação têm de estar registadas no ficheiro AndroidManifest.xml para que o sistema operativo as possa reconhecer e executar:

```
<activity
  android:name=".ui.MainActivity"
  android:label="@string/app_name">
    <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Para executar outra Activity B a partir da Activity A, usa-se o método **startActivity(intent)** na Activity A, onde "intent"é um objeto do tipo Intent. Esta classe representa a ação da Activity B que vai ser lançada e é

possível passar variáveis para este objeto para que seja imposto um estado inicial. Caso se espere um resultado da Activity B, usa-se **startActivityForResult(intent,id)** em que "id"é um inteiro que representa o pedido desta execução. O resultado da operação é depois devolvido no método **onActivityResult** da Activity A. O ciclo de vida da Activity pode ser visto na Figura 2.3.

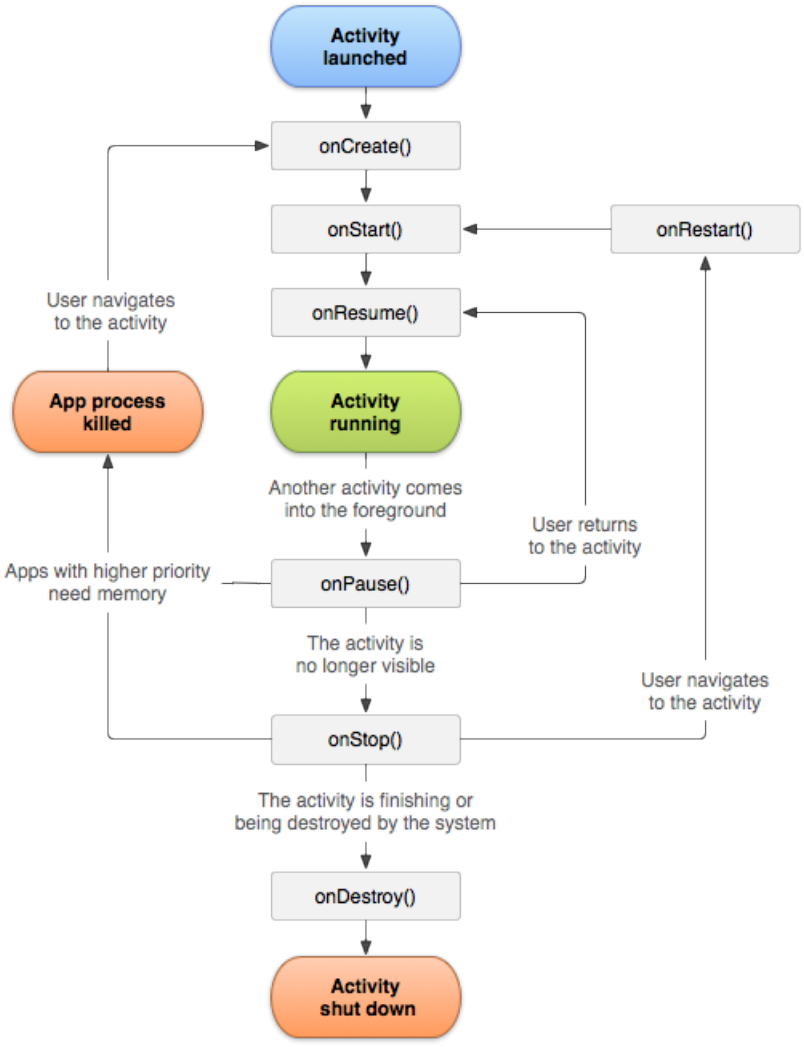


Figura 2.3: Ciclo de vida de uma Activity [10].

De forma resumida, os eventos mais importantes são:

- onCreate(Bundle savedInstanceState) - Quando a Activity é criada pela primeira vez ou reconstruída. O Bundle contém o estado guardado e é nulo da primeira vez que é executada. É neste método que se define o *layout* da interface gráfica e se atribuem as referências das Views.
- onStart() - Quando a Activity se encontra visível para o utilizador.
- onPause() - Quando a Activity muda para segundo plano.
- onSaveInstanceState(Bundle outState) - A Activity guarda o seu estado caso futuramente seja destruída.
- onStop() - Quando a Activity deixa de estar visível para o utilizador.
- onDestroy() - Quando a Activity é destruída devido a poupança de recursos do sistema ou quando o utilizador a termina explicitamente.

2.2.2 Fragment

Um Fragment representa uma porção do ecrã de uma Activity. São usados para separar diferentes partes da interface gráfica e reduzir as responsabilidades da Activity, simplificando o código necessário para desenvolver o ecrã em questão.

A Figura 2.4 contém dois exemplos em que se usam dois Fragments: A e B.

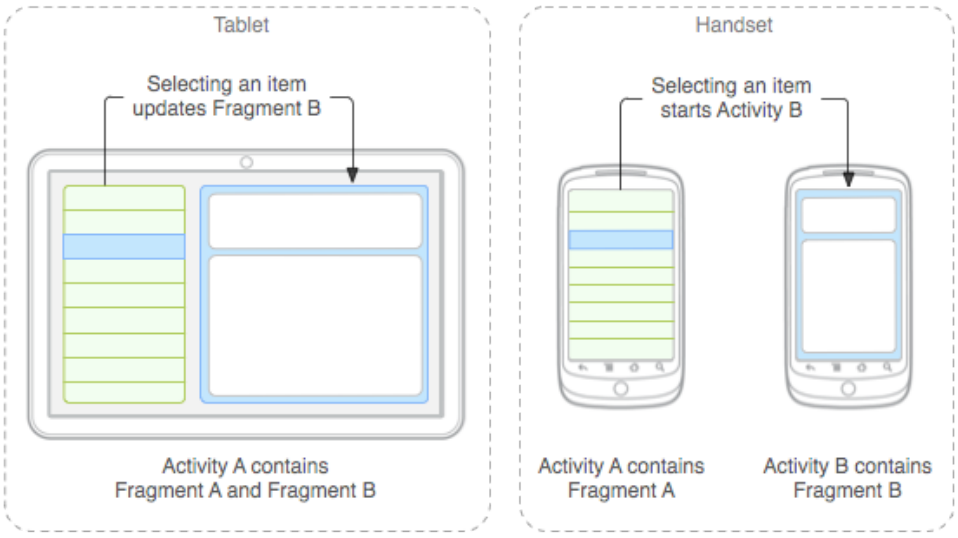


Figura 2.4: Comparação entre o uso de Fragments em *tablets* e *smartphones* [11]

O Fragment A representa uma lista de elementos e o Fragment B representa o conteúdo do elemento selecionado. Num *tablet*, como o ecrã é maior, podemos optimizar a interface para mostrar a lista e o detalhe ao mesmo tempo, usando para isso uma Activity com dois Fragments ativos ao mesmo tempo. Num smartphone, o que fazemos é substituir o Fragment A pelo Fragment B quando se clica num elemento da lista.

Os Fragments têm o seu próprio ciclo de vida, gerido pela Activity que os adiciona. Este ciclo de vida pode ser visto na Figura 2.5.

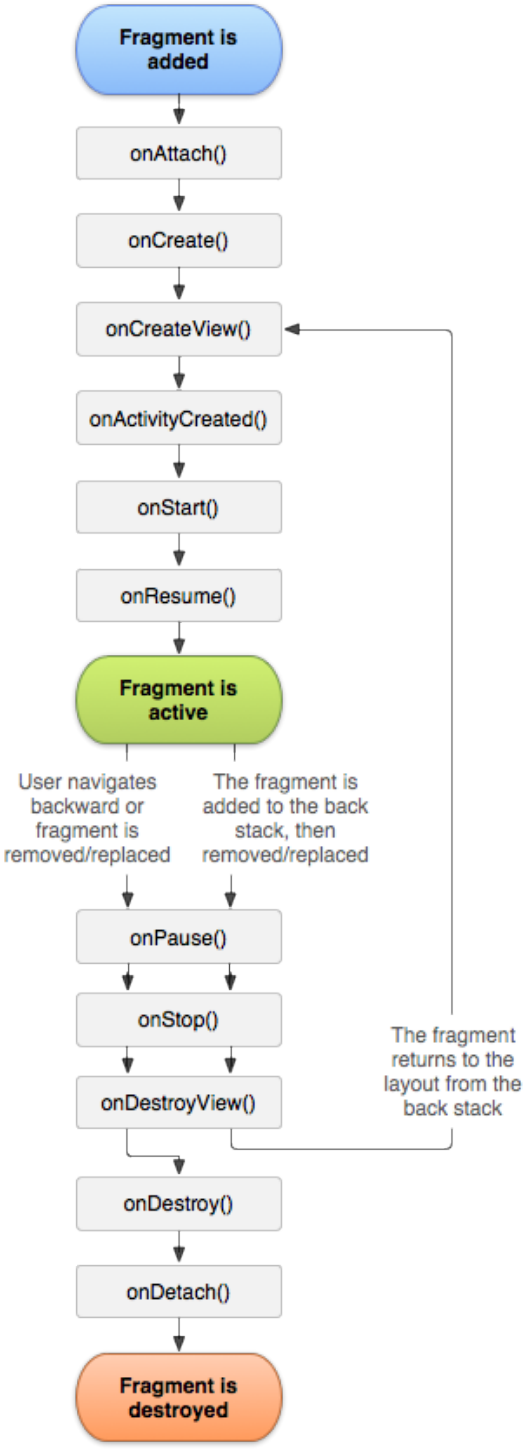


Figura 2.5: Ciclo de vida de um Fragment [11]

O ciclo de vida é bastante semelhante ao de uma Activity. Os eventos que a Activity não tem e que se justificam mencionar são:

- onAttach - Quando o Fragment é adicionado à Activity.
- onCreateView - Quando a View do Fragment é criada. Este método devolve uma View que será o Layout da interface gráfica.
- onDestroyView - Quando a View do Fragment é destruída.
- onDetach - Quando o Fragment é removido da Activity.

2.2.3 Service

Um Service representa um processo que é executado em plano de fundo no *smartphone*. É utilizado tipicamente para tarefas que demorem longos períodos de tempo e com as quais o utilizador não precise de interagir diretamente. Existem três tipos de Services:

- Foreground - serviço que mostra uma notificação ao utilizador a descrever a tarefa que está a executar. É normalmente utilizado quando se fazem transferências de ficheiros ou se reproduz música.
- Background - serviço que é executado sem que o utilizador se aperceba. Exemplo: serviço de atualização de localização.
- Bound - serviço que é utilizado para criar uma comunicação semelhante a cliente e servidor, em que o serviço atua como servidor e o cliente é a aplicação que utiliza o serviço. Exemplo: serviço de pagamentos do Google Play. As aplicações podem ligar-se a este serviço e fazer pedidos.

Um Service, à semelhança da Activity e do Fragment, também tem um ciclo de vida, que pode ser visto na Figura 2.6.

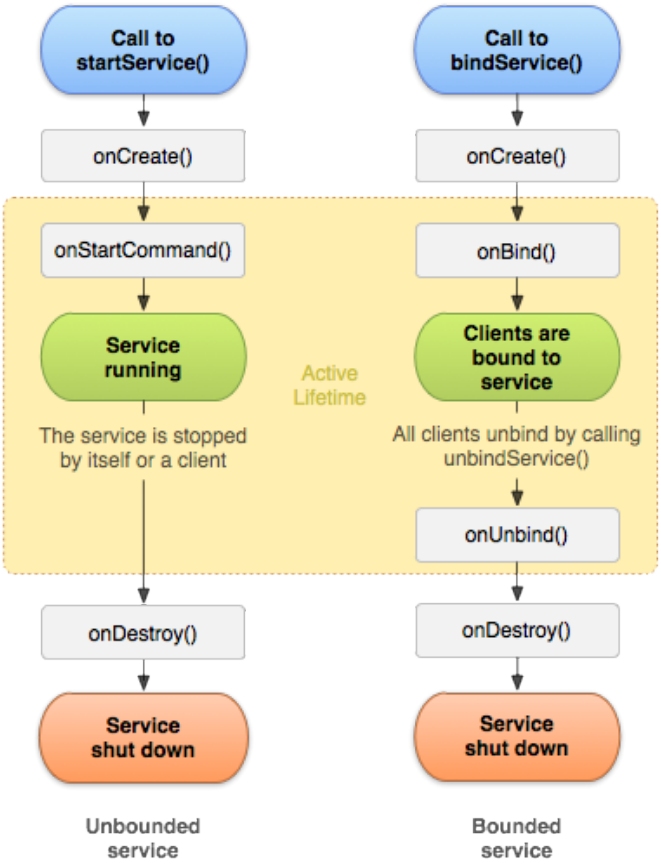


Figura 2.6: Ciclo de vida de um Service [12]

Na parte esquerda da figura, temos o ciclo de vida para Services do tipo Foreground e Background. Na direita, temos o ciclo de vida para Services Bound.

2.2.4 Arquitetura de uma Aplicação

Atualmente, o mais comum é desenvolver-se uma aplicação movendo certas responsabilidades das Activities e Fragments para outras classes, de forma

a facilitar o desenvolvimento e aumentar a testabilidade do código. Para isso, a comunidade de desenvolvimento para Android chegou a uma espécie de consenso e atualmente o padrão Model-View-Presenter (MVP) é o tipicamente mais utilizado para o desenvolvimento de aplicações. A Figura 2.7 explica sucintamente em que consiste este padrão de arquitetura. O objetivo deste padrão é reduzir a complexidade do código nas Activities e Fragments e torná-las o mais simples possível.

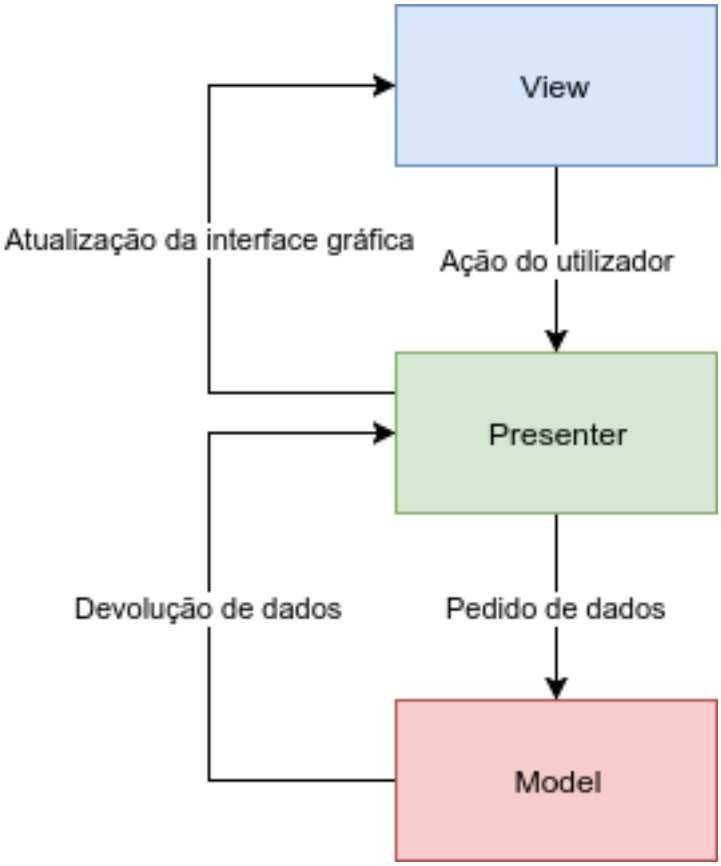


Figura 2.7: Demonstração do padrão MVP.

A View, em MVP, representa a interface gráfica, que pode ser uma Activity, um Fragment ou Dialog. A responsabilidade de uma View deve ser de

apenas gerir a interface gráfica e de delegar eventos ao Presenter.

O Presenter é uma classe simples em Java que trata dos eventos recebidos pela View e os processa, invocando o Model caso seja necessário.

O Model é um grupo de classes responsáveis pelos dados ou lógica da aplicação. Normalmente o Model é usado com o *Repository Pattern* (padrão de repositórios), que pode ser visto na Figura 2.8.

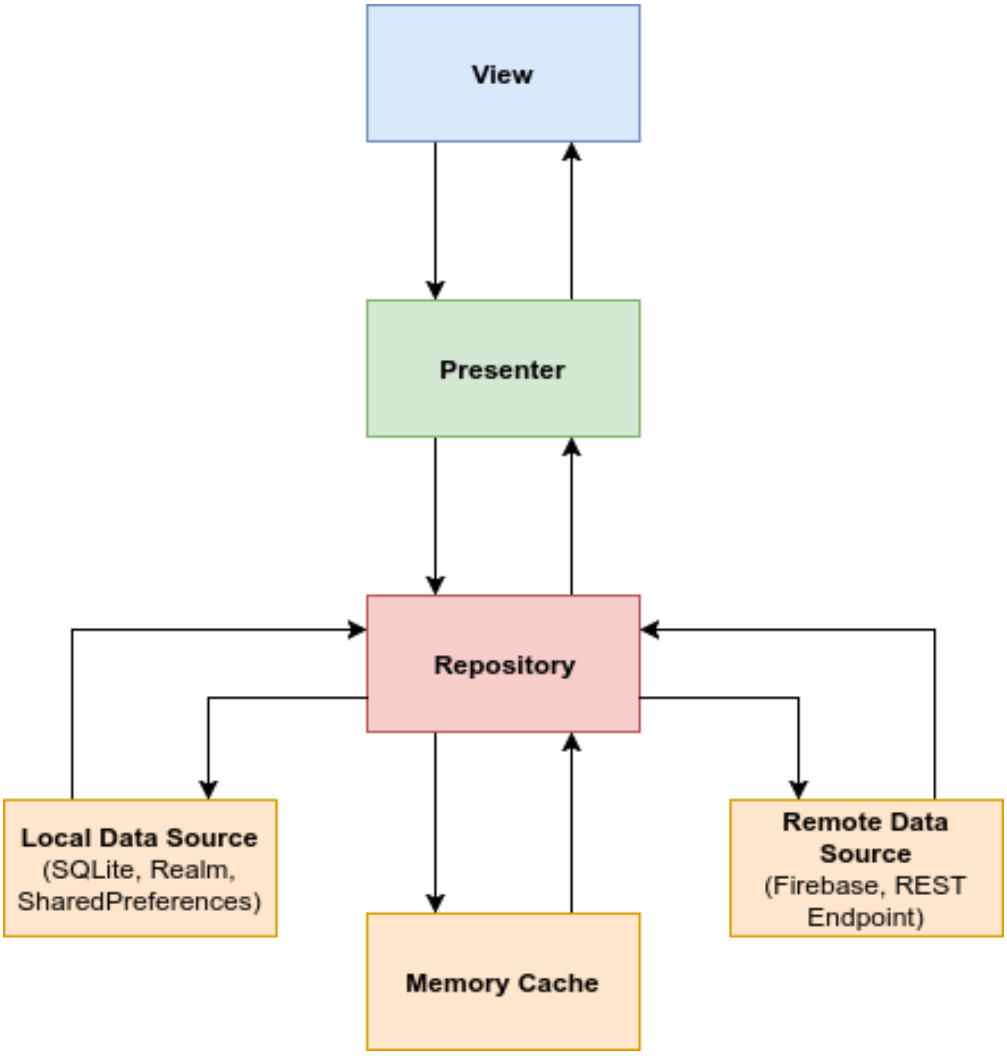


Figura 2.8: Padrão de repositórios em conjunto com MVP.

O objetivo deste padrão é abstrair a forma como o Presenter obtém os dados de forma a simplificar o código da aplicação. Um exemplo bastante comum numa aplicação é o seguinte: o utilizador pede uma lista de dados, a aplicação descarrega esses dados, guarda-os localmente no telemóvel (numa base de dados ou ficheiro) e mostra esses dados ao utilizador. Da próxima vez que o utilizador pedir essa lista, podemos carregar os dados da base de dados local em vez de os descarregar da Internet de novo. O repositório neste caso é quem decide quando é que se descarregam os dados da Internet ou da base de dados local.

2.3 Serviços de Cloud

Nesta secção são abordados diferentes serviços de Cloud que são utilizados no âmbito da IoT. São descritas as caraterísticas de cada serviço e apresentada uma comparação entre todos de forma a justificar a escolha que foi tomada: optar pelo Firebase.

2.3.1 Microsoft Azure

O Microsoft Azure inclui os seguintes produtos relevantes para o contexto desta dissertação:

- Hub IoT - sistema que permite a integração e gestão automática de dispositivos IoT.
- Azure Cosmos DB - sistema de base de dados distribuída.
- Funções - sistema que permite definir reações eventos (alterações na base de dados, inserção de ficheiros) e desencadear outras ações.
- Hubs de Notificação - sistema de gestão de notificações.

2.3.1.1 Hub IoT

O Hub IoT é um serviço que permite gerir de forma automática conjuntos de dispositivos de IoT. Na Figura 2.9 podemos ver o papel deste serviço numa arquitetura de IoT que envolva um conjunto de dispositivos e um *backend* que processa os dados recolhidos.

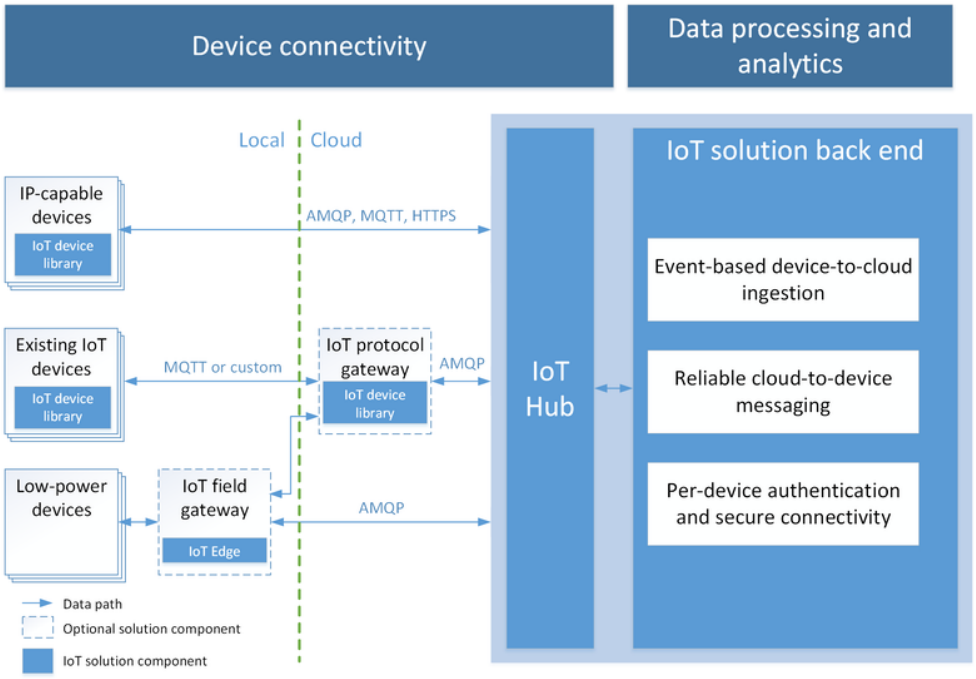


Figura 2.9: Arquitetura de um sistema de IoT com o Hub IoT [13].

Cada dispositivo é autenticado para garantir comunicações seguras com o Hub IoT. Para a integração ser possível, é necessário desenvolver um programa para os dispositivos usando o SDK oficial [14].

Felizmente, existe um tipo deste serviço que é gratuito, para ser usado durante uma fase inicial de desenvolvimento. Os preços podem ser vistos na Tabela 2.1.

Tabela 2.1: Tabela de preços do Hub IoT [15].

Tipo de edição	Preço mensal	Mensagens por dia	Tamanho das mensagens
Gratuita	0€	8.000	0,5 KB
S1	42,17€	400.000	4KB
S2	421,65€	6.000.000	4KB
S3	4216,50€	300.000.000	4KB

Se assumirmos que temos dispositivos a enviar mensagens a cada minuto, podemos ter:

$\frac{8000}{60\times24} \approx 5$ dispositivos ligados sem atingir o limite gratuito.

No entanto, se tivermos dispositivos a enviar mensagens a cada segundo, já é necessário optar pela edição S1:

$\frac{400000}{3600\times24} \approx 4$ dispositivos.

Assim podemos observar que para um grande volume de dados e dispositivos, torna-se excessivamente caro usar o Hub IoT. Imaginando 10.000 dispositivos ligados e a enviar mensagens a cada segundo:

$\frac{10000\times3600\times24}{300000000} \approx 3$ instâncias S3, o que equivale a 12.649,5€ por mês.

2.3.1.2 Funções

O serviço de Funções do Microsoft Azure permite desenvolver programas que são executados por certos eventos, sem ser necessária a manutenção e configuração de servidores. As Funções podem ser escritas em CSharp, JavaScript ou FSharp e podem ser executadas com base nos seguintes eventos:

- Pedidos HTTP - quando se faz um GET para um *endpoint* registado no Azure.

- Temporizadores - quando passa um certo tempo definido pelo utilizador.
- Azure Cosmos DB - quando um objeto da base de dados é alterado.
- Ficheiros - quando são inseridos ou removidos ficheiros do armazenamento no Azure.

Os preços podem ser vistos na Tabela 2.2.

Tabela 2.2: Tabela de preços das Funções Azure [16].

Medidor	Preço mensal	Quota mensal gratuita
Tempo de execução	0,000014€/GB·s	400.000 GB·s
Total de execuções	0,169€/milhão de execuções	1 milhão de execuções

As funções são faturadas com base no consumo de memória e mede-se em segundos de gigabytes (GB·s). Para isso, multiplica-se o número de execuções pelo tempo de execução e de seguida multiplica-se pelo consumo médio de memória em gigabytes.

Continuando os exemplos anteriores, imaginemos que temos 4 dispositivos a enviar uma mensagem por segundo e que executámos sempre uma função quando se recebe a mensagem. Assumiremos também que a função demora meio segundo a executar e que tem um consumo médio de 1024 MB (1 GB) de memória.

Assim, por cada segundo, temos 4 execuções no total:

$4 \times 3600 \times 24 \times 30 = 10.368.000$ execuções por mês.

Depois, para ter o consumo de tempo, multiplica-se pela duração da função:

$10.368.000 \times 0,5 = 5.184.000$ segundos.

Resta só multiplicar pelo consumo médio de memória:

1 GB$\times\frac{1024MB}{1024MB} \times 5.184.000 = 5.184.000$ GB·s

Retira-se o consumo de tempo gratuito:

$5.184.000 - 400.000 = 4.784.000$ GB·s

Retira-se também o consumo de execuções gratuito:

$10.368.000 - 1.000.000 = 9.368.000$ execuções por mês

Assim, o consumo total das funções é de:

$4.784.000 \times 0,000014 + \frac{9.368.000}{1.000.000} \times 0,169 \approx 68{,}56$€ por mês.

Se refizermos os cálculos para 10.000 dispositivos, temos:

$10.000 \times 3.600 \times 24 \times 30 - 1.000.000 = 25.920.000.000$ execuções por mês

$25.920.000.000 \times 0,5 - 400.000 = 12.959.600.000$ GB·s

$12.959.600.000 \times 0,000014 + \frac{25.919.000.000}{1.000.000} \times 0,169 \approx 185.814{,}71$€ por mês.

2.3.1.3 Azure Cosmos DB

O Azure Cosmos DB é um serviço de base de dados que oferece uma distribuição global com latências bastante reduzidas. A base de dados é do tipo não relacional e inclui suporte para o DocumentDB e MongoDB, dois tipos de bases de dados NoSQL. Ao ser não relacional, reduz tempo de desenvolvimento, visto que só temos de guardar objetos e não nos preocupar com a gestão de tabelas e as suas relações. É um serviço que escala automaticamente, reduzindo também a responsabilidade do utilizador em adquirir mais servidores e gerir o espaço manualmente.

A faturação é calculada com base nas Request Units (RU) (unidades de pedido) por segundo e no espaço usado para o armazenamento dos dados. Os preços podem ser vistos na Tabela 2.3.

Para o cálculo de custos, iremos assumir que cada dispositivo consome uma RU por segundo e que existem 100 KB de dados por dispositivo na base

Tabela 2.3: Tabela de preços da Azure Cosmos DB [17].

Medidor	Unidade	Preço
Armazenamento SSD	GB	0,2109€/GB
Débito	100 RU/segundo	0,0068€/hora

de dados. Assim, para 4 dispositivos, temos:

$\frac{4\times100\times1024}{1024^3} \times 0,2109 = 0,000080452$€. O custo de armazenamento não chega a 1 cêntimo, neste caso.

Como precisamos apenas de 4 RU/s, o custo de reserva do débito é de:

$0,0068 \times 24 \times 30 \approx 4,90$€.

Para 10.000 dispositivos, vamos assumir que apenas 1.000 dispositivos é que escrevem a cada segundo e que precisamos de 1.000 RU/s. Como a faturação é feita por cada 100 RU/s, faz-se:

$\frac{1.000}{100} \times 0,0068 \times 24 \times 30 = 48,96$€.

O custo de armazenamento neste caso é de:

$\frac{10.000\times100\times1024}{1024^3} \times 0,2109 \approx 0,20$€.

2.3.1.4 Hubs de Notificação

Os Hubs de Notificação são uma plataforma que permite gerir de forma automática o envio de notificações para inúmeros dispositivos ligados à Internet. Funciona com iOS, Android, Windows e Kindle, usando os serviços de notificações *push* das próprias empresas. As notificações *push* são notificações que servem para alertar os utilizadores de certos eventos que sejam relevantes no contexto da aplicação instalada. No caso desta dissertação, as notificações são enviadas para alertarem o utilizador acerca da carga da bateria do seu veículo.

Gerir a infraestrutura de um sistema de notificações é uma tarefa bastante

complicada. É necessário implementar um sistema de registo dos identificadores dos dispositivos ativos, validar se o identificador ainda é válido e se não for, pedir um novo. Tudo isto envolve consumo de recursos e para milhões de dispositivos a gestão torna-se um desafio.

Na Figura 2.10 podemos ver uma arquitetura de um sistema de notificações. Os dispositivos contactam o Platform Notification Service (PNS) para obterem um identificador único que será usado para receberem notificações. Depois, enviam este identificador para o *back-end* da aplicação, que depois o usará para enviar notificações quando for necessário.

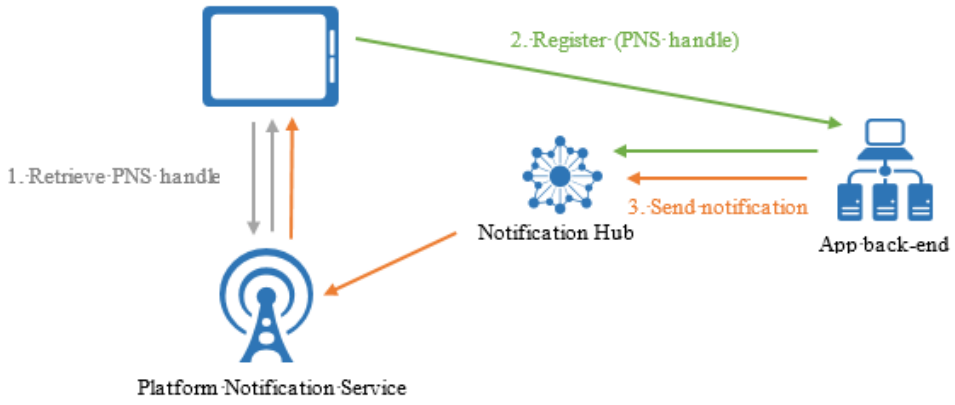


Figura 2.10: Arquitetura de um sistema de notificações com o Hub de Notificação [18].

Algumas das vantagens de incluir o Hub de Notificações são:

- Suporte a múltiplas plataformas (Android, iOS, Windows Mobile).
- Possibilidade de atribuir etiquetas aos dispositivos (idioma, localização) para garantir a entrega um certo público alvo.

- Suporte a múltiplas linguagens para a integração com qualquer *back-end*

Este serviço é gratuito até 1 milhão de notificações. A partir do primeiro milhão, cada milhão de notificações extra custa 0,844€.

2.3.2 Amazon Web Services

A Amazon disponibiliza serviços semelhantes aos do Microsoft Azure. Nesta secção serão descritos esses serviços e apresentados os preços.

2.3.2.1 AWS IoT

O AWS IoT é composto por uma série de ferramentas que possibilitam a integração de sensores com os serviços da Amazon. É disponibilizado um Software Development Kit (SDK), AWS IoT Device SDK, para facilitar a configuração e ligação do *hardware* às aplicações desenvolvidas. Este SDK possibilita a ligação dos dispositivos ao AWS IoT, de forma segura, com autenticação por cada dispositivo e a troca de mensagens com os protocolos Message Queue Telemetry Transport (MQTT), HTTP ou WebSockets.

Na Figura 2.11 podemos ver as funcionalidades do AWS IoT.

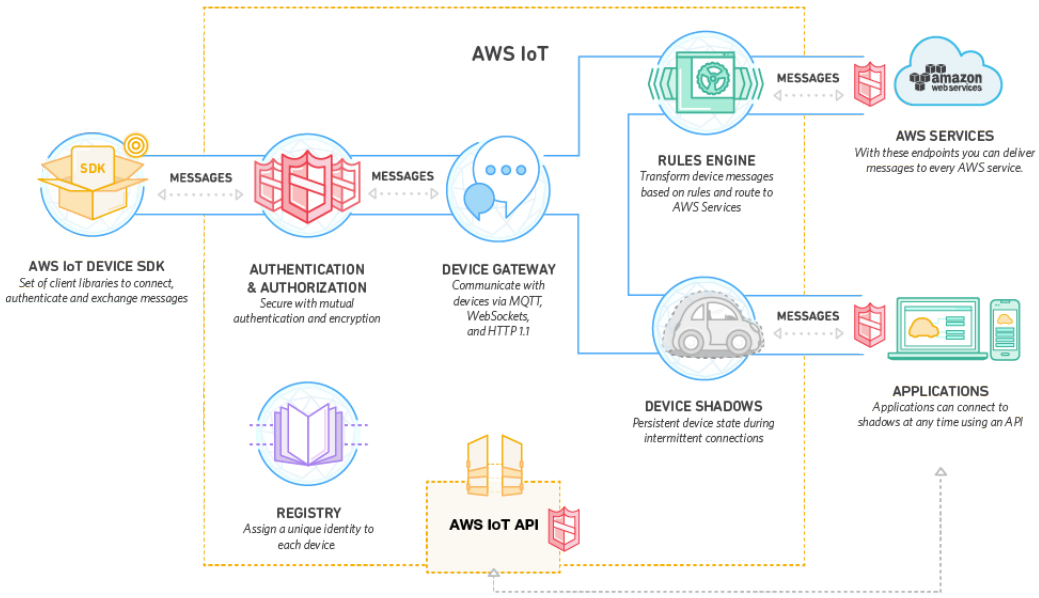


Figura 2.11: Arquitetura de um sistema IoT com o AWS IoT [19].

O custo do AWS IoT é de 5$ por cada milhão de mensagens, se a região escolhida for europeia. O preço total inclui as mensagens enviadas para o AWS IoT e as mensagens recebidas pelos dispositivos.

Retomando o exemplo dos 10.000 dispositivos, se cada um enviar uma mensagem segundo a segundo:

$10.000 \times 3600 \times 24 \times 30 = 25.920.000.000$ mensagens por mês

$\frac{25.920.000.000}{1.000.000} \times 5 = 129.600\$$ por mês (cerca de 110.160€ com a taxa de câmbio atual de 0.85).

2.3.2.2 AWS Lambda

O AWS Lambda é o serviço equivalente às Funções do Azure. O AWS Lambda permite executar código (JavaScript, Python, Java ou C#) sem a preocupação de gerir servidores, pagando-se apenas o tempo de computação

necessário para a execução da função. Um Lambda (Função) pode ser invocado por outros serviços da Amazon ou por aplicações móveis e até *websites*. Na Figura 2.12 podemos ver um resumo de como funciona o AWS Lambda.

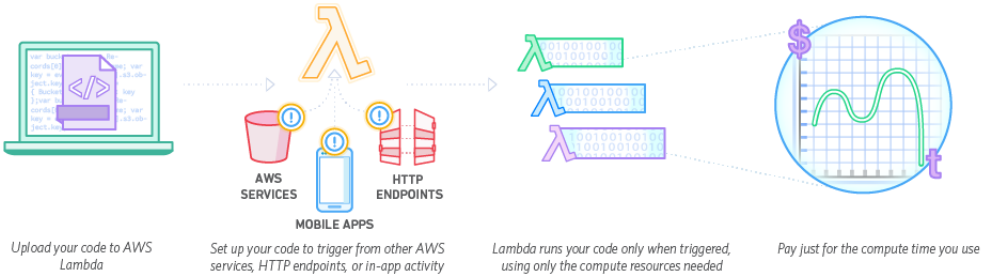


Figura 2.12: Explicação geral de como funciona o AWS Lambda [20].

O código enviado para o AWS Lambda é invocado com certos eventos dos serviços da AWS ou pedidos de aplicações móveis.

A faturação também é idêntica às Funções do Azure, com a diferença da taxação a cada 100 milissegundos. As quotas gratuitas são iguais entre os dois serviços: 1 milhão de execuções e 400.000 GB·s. Cada milhão de execuções extra também custa 0,169€. Na Tabela 2.4 estão descritos os preços para alguns valores de memória escolhidos (à taxa de 1$ = 0,85€).

Tabela 2.4: Tabela de preços do AWS Lambda [21].

Memória (MB)	Quota gratuita (GB·s)	Preço (€)
128	3.200.000	0,00000177
256	1.600.000	0,00000354
512	800.000	0,00000709
1024	400.000	0,00001417
1536	266.667	0,00002126

Para 1 GB alocado de memória, 10.000 execuções por segundo e duração de 0,5 segundos durante um mês:

$10000 \times 3600 \times 24 \times 30 = 25.920.000.000$ execuções por mês

A duração em GB·s é de:

$25.920.000.000 \times 0,5 - 400.000 = 12.959.600.000$ GB·s

O custo total é de: $12.959.600.000 \times 0,00001417 + \frac{25.919.000.000}{1.000.000} \times 0,169 \approx 188.017,84$€ por mês.

2.3.2.3 Amazon DynamoDB

A Amazon DynamoDB é uma base de dados NoSQL disponível na Cloud e automaticamente escalável, sem necessidade de gestão do utilizador. É um produto concorrente à Azure Cosmos DB que oferece preços também atrativos e integra-se facilmente com o AWS Lambda para a criação de aplicações de forma mais simples.

A faturação é calculada com base em unidades de capacidade de gravação (WCU), unidades de capacidade de leitura (RCU) e armazenamento utilizado. Os preços podem ser vistos na Tabela 2.5.

Tabela 2.5: Tabela de preços do AWS DynamoDB [22].

Tipo de recurso	Quota gratuita	Preço (€)
Escrita	25 WCU	0,4/WCU
Leitura	25 RCU	0,08/RCU
Armazenamento	25 GB	0,21/GB

Com 10.000 dispositivos a consumir 1.000 WCUs e cada um com 100 KB de dados armazenados, temos os seguintes preços:

$\frac{10.000 \times 100 \times 1024}{1024^3} \approx 0.95$GB. Como não chega aos 25 GB, o armazenamento é gratuito e paga-se apenas $(1.000 - 25) \times 0,4 = 390$€ de consumo de escrita.

2.3.2.4 Amazon SNS

O Amazon Simple Notification Service (SNS) é um serviço que gere notificações. Com este serviço, é possível enviar mensagens para um vasto número de subscritores e de várias formas: notificações push, email ou SMS. Existe também integração com outros serviços do AWS. Na Figura 2.13 podemos ver a arquitetura do sistema de alertas de recursos utilizado pela Amazon (AWS Limit Monitor).

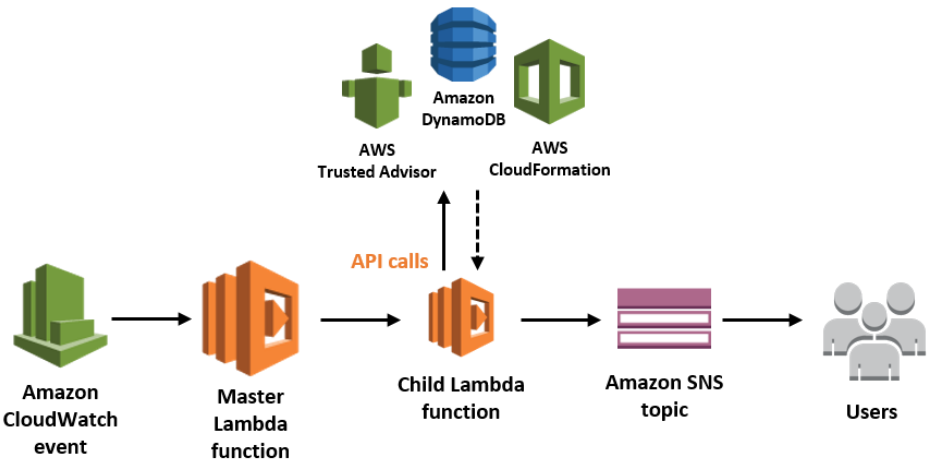


Figura 2.13: Exemplo de aplicação do Amazon SNS [23].

Este sistema usa o AWS Lambda para enviar uma notificação através do Amazon SNS. A função do AWS Lambda é invocada assim que o CloudWatch (serviço que monitoriza os recursos do AWS) relata uma utilização superior a uma dada percentagem. Depois, o Amazon SNS envia um email ao utilizador sobre a utilização estar a chegar ao limite.

O custo das notificações é gratuito até 1 milhão de notificações por mês. Depois, cada milhão extra custa 0,425€, cerca de metade do preço do Hub

de Notificações do Azure.

2.3.3 Firebase

O Firebase é um Backend as a Service (BaaS) com produtos desenvolvidos para facilitar o desenvolvimento e manutenção da infraestrutura de aplicações e *websites*. Inclui serviços semelhantes aos referidos anteriormente e possuiu a vantagem de ter um plano gratuito para todas as funcionalidades, sem a necessidade de adicionar um cartão de crédito. Existem 3 planos de pagamento:

- Spark - gratuito.
- Flame - 21.25€ (25$) por mês.
- Blaze - conforme o uso. Pode ser mais barato que o Flame se forem usados poucos recursos.

Os preços simulados na subsecções seguintes têm em conta a seleção do plano Blaze. Com este plano, os preços podem não atingir os 21.25€, visto que se paga apenas o que é usado. O plano Flame serve apenas para controlar os custos e evitar faturas elevadas. No entanto, para projetar um sistema que escale para milhões de dispositivos como nesta dissertação, o plano Blaze é a única escolha que faz sentido.

2.3.3.1 Autenticação

O Firebase inclui suporte gratuito para os seguintes métodos de autenticação:

- Email e senha.

- *Login* da Google.
- *Login* do Facebook.
- *Login* do Github.
- *Login* do Twitter.
- Customizável (com *tokens* geridos por outro serviço).

A integração com estes métodos de autenticação requer apenas algumas linhas de código, o que simplifica o desenvolvimento de aplicações e a manutenção a longo prazo. Os métodos de *login* de serviços externos (Google, Facebook, etc) são geridos por *access tokens*. Para isso, é necessário configurar acesso à aplicação que está a ser desenvolvida nas consolas de gestão da Google ou do Facebook.

2.3.3.2 Base de Dados em Tempo Real

A base de dados em tempo real é um dos principais serviços do Firebase. É possível escutar por alterações em certos objetos na base de dados e ser notificado em tempo real das alterações, o que torna bastante útil para aplicações no contexto de IoT, visto que a latência é bastante importante. A base de dados não é relacional e os objetos são guardados em formato JavaScript Object Notation (JSON). Existe também suporte *offline* para que a experiência do utilizador não se degrade caso as condições de ligação à rede não sejam as melhores. Neste caso, quando a aplicação não consegue obter os dados mais recentes a partir da Internet, são devolvidos os dados na *cache* de forma automática.

Em relação aos preços, para 10.000 dispositivos que requerem 100 KB de dados, já sabemos que não chega a 1 GB (ronda os 0,95 GB). Assim, não se

paga o armazenamento neste caso, visto ser gratuito até 1 GB. Assumindo que estes 10.000 dispositivos escrevem por cada segundo mensagens de 0,5 KB, temos um consumo total de:

$\frac{10.000 \times 0{,}5 \times 1024 \times 3600 \times 24 \times 30}{1024^3} \approx 12360$ GB de volume de dados enviado.

Como cada GB custa 0,85€ e existem 20 GB gratuitos, temos um preço total de 10.489€.

2.3.3.3 Funções

As Funções do Firebase cumprem o mesmo propósito que o AWS Lambda, mas só podem ser desenvolvidas em JavaScript. Podem reagir a eventos de autenticação, alterações na base de dados, alterações nos ficheiros alojados e a eventos do Google Analytics.

Cada milhão de invocações custa 0,34€ e as primeiras 2 milhões são gratuitas. No entanto, ao contrário do AWS Lambda e do Azure, também se pagam os ciclos de relógio (medidos em GHz·s). Com o cenário dos 10.000 dispositivos temos 25.920.000.000 execuções por mês, 12.959.600.000 GB·s e 12.959.600.000 GHz·s.

Assim, o preço total seria de:

$0,002125 \times \frac{12.959.600.000}{1000} + 0,0085 \times \frac{12.959.800.000}{1000} + \frac{25.918.000.000}{1.000.000} \times 0,34 = 146.509,57$€

2.3.3.4 Notificações

O serviço de notificações do Firebase chama-se Firebase Cloud Messaging (FCM) e é completamente gratuito. Podem ser enviadas dois tipos de mensagens através da consola do Firebase e dos *endpoints* HTTP:

- Mensagens de notificação: Pode ser definido um título, legenda, ícone e o público alvo através de atributos definidos no Firebase (idioma,

localização, etc).

- Mensagens de dados: mensagens com pares de chave e valor.

2.3.4 Comparações

Nesta subsecção serão apresentadas comparações de preços entre os três serviços descritos anteriormente. Os dados das comparações são os seguintes:

- Mensagens de 0,5 KB enviadas dos dispositivos para a Cloud.
- 100 KB de armazenamento por dispositivo.
- Não serão contabilizadas as leituras de dados.

Num primeiro cenário, de 10.000 dispositivos a enviar mensagens a cada segundo, temos os seguintes preços mensais:

Tabela 2.6: Preços dos vários serviços de Cloud para 10.000 dispositivos a enviar mensagens a cada segundo.

Cloud	Plataforma IoT	Base de dados	Funções	Mensalidade
Azure	12.649,5€	49,91€	185.814,71€	198.954€
AWS	110.160€	390€	188.017,84€	302.167,84€
Firebase	Não tem	10.489€	146.509,57€	156.996€

Apesar do Firebase não incluir uma plataforma de IoT, continua a ser o serviço mais barato, devido aos preços das funções. Apesar do seu custo de base de dados ser quase duas vezes superior ao da Amazon e vinte vezes superior ao do Azure, importa realçar que é um serviço diferente dos restantes. Inclui suporte para alterações de dados em tempo real através de um

SDK bastante fácil de integrar numa aplicação, o que não existe nos serviços concorrentes.

Num segundo cenário, com 100.000 de dispositivos a enviar mensagens a cada minuto, temos os seguintes preços:

Tabela 2.7: Preços dos vários serviços de Cloud para 100.000 dispositivos a enviar mensagens a cada minuto.

Cloud	Plataforma IoT	Base de dados	Funções	Mensalidade
Azure	4.216,5 €	499.14€	30.964,31 €	35.679,94€
AWS	18.360 €	3.990 €	31.331,44 €	53.681,44€
Firebase	Não tem	1.733,95 €	24.415,57 €	26.149,52€

Foi possível descer os preços em cerca de 80% para o Firebase, aumentando o número de dispositivos em 10 vezes e reduzindo o número de mensagens em 60 vezes.

A plataforma de IoT garantiria a autenticação do *gateway* à base de dados e uma melhor gestão dos dispositivos, no entanto, como o Firebase oferece integração com diversos serviços de autenticação de forma gratuita, não há necessidade de incluir este serviço. Assim, como não é necessária uma plataforma de IoT para a realização desta dissertação, a escolha do Firebase como serviço de *cloud* desta dissertação acabou por ser a mais vantajosa em termos de preços.

Para um sistema IoT que necessite de uma plataforma de IoT, a solução que tem mais em conta o preço é o Azure da Microsoft. A base de dados, as funções e a plataforma de IoT são mais acessíveis em termos de preço que os concorrentes da AWS.

2.4 Trabalho relacionado

Nesta secção serão mencionados exemplos de projetos na área de IoT e que são relevantes para o contexto desta dissertação.

Em [24], foi desenvolvido um sistema de antirroubo para veículos. A deteção de roubo é feita com sensores de vibração que atuam quando existe uma abertura forçada do carro e de sensores infravermelhos que detetam a presença de alguém no lugar do condutor. Estes sensores só estão ativos enquanto uma *tag* RFID, presente na chave do carro, estiver longe do leitor que se encontra dentro. O fluxo deste sistema pode ser visto na Figura 2.14.

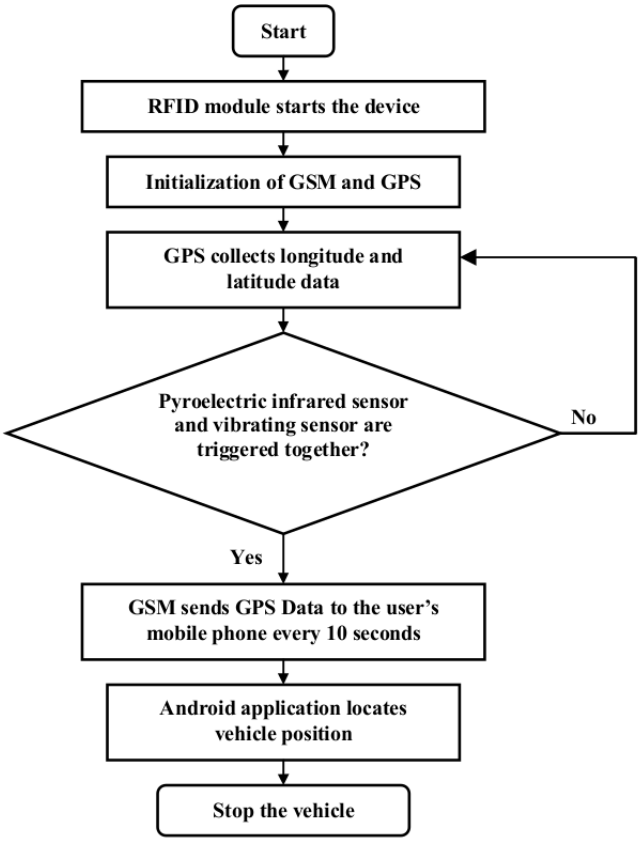


Figura 2.14: Fluxo do sistema de antiroubo de veículos [24].

Caso o roubo seja detetado, é enviada uma SMS a cada 10 segundos que contém as coordenadas geográficas atuais, através de um módulo GSM instalado dentro do carro. Foi também desenvolvida uma aplicação Android que possibilita trancar ou destrancar o carro, ou até cortar o acesso ao combustível, enviando uma SMS para o módulo GSM.

Em [25], foi proposto um sistema de carregamento dinâmico de veículos elétricos para a integração com casas inteligentes. O sistema ativa ou desativa o carregamento conforme a carga de corrente na casa. Se a carga for alta, o sistema funciona de modo a que a bateria forneça energia à rede. Caso contrário, se a carga for baixa, o sistema funciona de modo a fornecer energia à bateria, conforme a carga disponível para o carregamento.

Em [26], foi desenvolvido um sistema de gestão de lugares de estacionamento. Existem sensores nos lugares de estacionamento que se ligam remotamente a uma estação base através do protocolo LoRaWAN. Por sua vez, a estação base envia os dados para uma Cloud de forma a que aplicações possam consultar o estado dos lugares quase em tempo real. O sistema foi implementado no Dubai.

Em [27], foi desenvolvido um sistema de monitorização de veículos com sincronização para uma *cloud* da IBM, a IBM Bluemix. Os dados do carro, tais como temperatura do motor, níveis de combustível, velocidade, são recolhidos a partir do On-board Diagnostic System (OBD-II) e enviados para um *smartphone*, que atua como *gateway*, via Bluetooth. O fluxograma deste sistema pode ser visto na Figura 2.15.

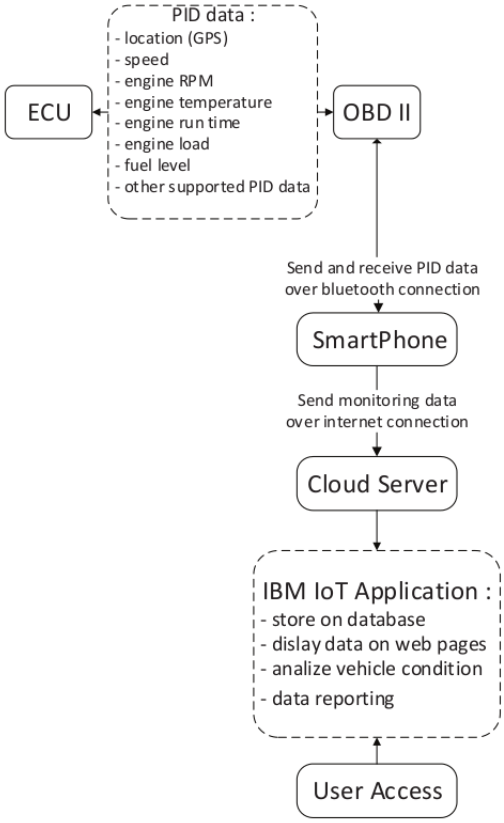


Figura 2.15: Fluxograma do sistema de monitorização de veículos [27].

Os utilizadores, depois de autenticados, podem consultar os dados do seu veículo através de uma página *web* na IBM BlueMix.

Em [28], foi desenvolvido um sistema de monitorização de colmeias. Como é sabido, as abelhas são responsáveis por grande parte da polinização de todas as plantas, mas tem havido um declínio no seu número. Como tal, surgiu a necessidade de investir na segurança da espécie para garantir que esta não se extinga. A Figura 2.16 representa a arquitetura do sistema.

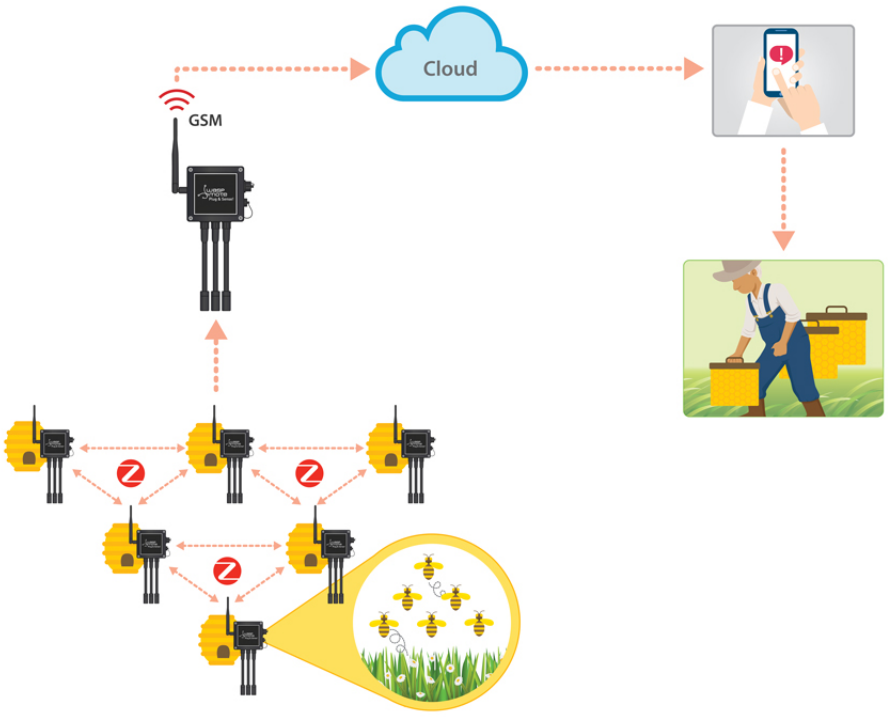


Figura 2.16: Sistema de monitorização remota de colmeias [28].

Existem sensores na colmeia que leem uma variedade de parâmetros: níveis de dióxido de carbono, oxigénio, temperatura, humidade, poluentes e até poeira. Os sensores comunicam com um *gateway* com o protocolo ZigBee e este comunica os resultados para uma *cloud* via GSM. Com os dados guardados e acessíveis *online*, é possível acompanhar o estado da colmeia em tempo real e estudar o comportamento das abelhas.

Em [29], foi desenvolvida uma aplicação que controla remotamente certos aparelhos em casa (ventoinha e luzes). A arquitetura é composta por uma aplicação cliente que usa o Google Assistant, um sensor que liga ao Android Things, um sistema operativo baseado no Android para dispositivos IoT, Google Home (dispositivo que funciona como *gateway*) e Firebase como

servidor. A Figura 2.17 mostra os dispositivos que são controlados em casa.

Figura 2.17: Sistema de controlo remoto de dispositivos com Android Things e Google Home [29].

É possível fazer perguntas ao assistente da Google sobre o estado das luzes e da ventoinha: "A ventoinha está ligada?". Ao fazer a pergunta, o sistema faz um pedido HTTP a um *endpoint* criado pelo desenvolvedor e devolve o estado atual.

Capítulo 3

Desenvolvimento do Sistema

Neste capítulo, serão descritas as implementações dos diferentes componentes da arquitetura do sistema, que pode ser vista na Figura 3.1.

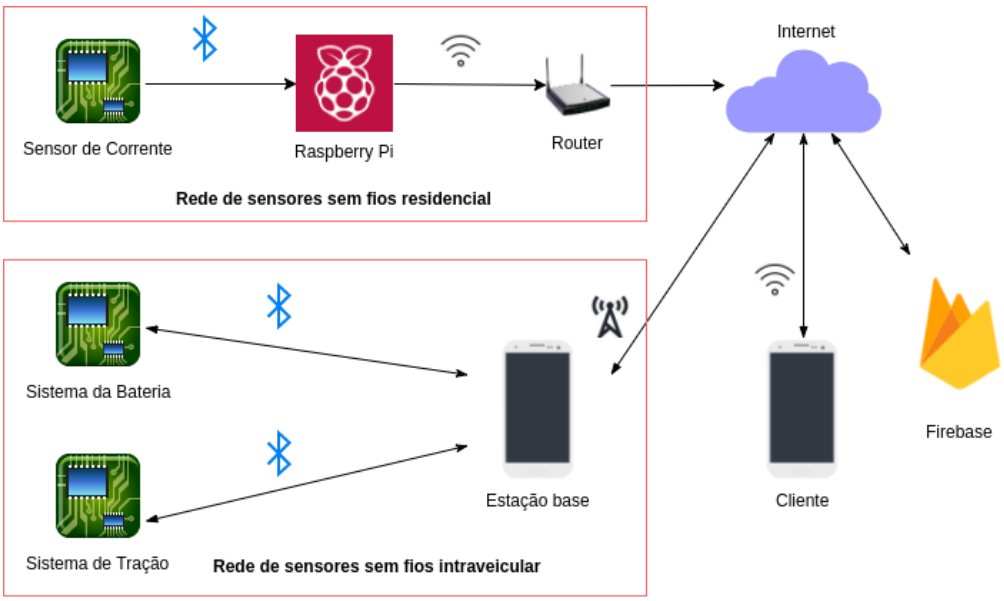


Figura 3.1: Arquitetura geral do sistema.

Na primeira secção, é descrita a configuração dos diferentes serviços do Firebase para a integração com o programa para o Raspberry Pi e para a

aplicação cliente. De seguida, é explicada a configuração dos dispositivos periféricos que enviam os dados dos sensores da bateria, sistema de tração e do sistema de corrente doméstica. Depois, é explicado como foi desenvolvido o programa para o Raspberry Pi, que sicroniza os dados de corrente para o Firebase. Por fim, na última secção, é explicado como foi desenvolvida a aplicação cliente.

3.1 Configuração do Firebase

Nesta secção é explicado como foram configurados os diferentes serviços do Firebase, com recurso a exemplos e código (no caso das funções). Antes da configuração, é necessário criar um projeto no Firebase que identifique a aplicação móvel.

3.1.1 Autenticação

O método escolhido para autenticar os utilizadores da aplicação foi o da conta da Google. Como todos os utilizadores de Android são obrigados a ter uma conta da Google para descarregarem aplicações e usar a combinação de email e senha nem sempre é conveniente, torna-se mais fácil entrar na aplicação com um simples botão que abre a lista de contas da Google no dispositivo.

Para ativar este método, é necessário ir à consola do Firebase, escolher o projeto da aplicação móvel e selecionar a secção "Authentication", que se encontra no painel lateral. Depois, no separador "Sign-in method", ativa-se a opção "Google".

Para os serviços do Google Play reconhecerem a aplicação que está a tentar usar o serviço de autenticação, é necessário adicionar a impressão digital

do certificado que é usado para assinar o ficheiro .apk. Esta impressão digital pode-se obter com a ferramenta keytool que está incluída na instalação do Java, ou através de uma opção no Android Studio. Para obter a impressão digital através do Android Studio, abre-se a janela do Gradle, escolhe-se o módulo que corresponde à aplicação e depois, "Tasks", "android" e "signingReport". Este comando devolve o seguinte no terminal:

```
Variant: debugAndroidTest
Config: debug
Store: /home/ruben/.android/debug.keystore
Alias: AndroidDebugKey
MD5: A9:CF:35:8F:38:60:B4:1D:25:6C:AE:0E:91:B0:6B:4A
SHA1: CF:B1:7D:39:95:15:8F:34:03:86:28:C9:2F:92:09:73
:C7:25:2C:63
Valid until: Thursday, March 7, 2047
```

No Firebase adiciona-se a impressão digital SHA1 nas definições da aplicação. Depois, na aplicação, é necessário pedir um *access token* ao serviço da Google e usá-lo para autenticar com o Firebase. Esta parte será explicada com melhor detalhe na secção de desenvolvimento da aplicação.

3.1.2 Base de Dados

A estrutura da base de dados desta dissertação resume-se a um único ficheiro JSON que contém coleções de objetos. O primeiro passo para estruturar este foi identificar os dados que são necessários recolher:

- Dados do utilizador: definições das notificações.
- Corrente do sistema de carregamento residencial.
- Sistema de bateria.
- Sistema de tração.

- Localização.

A estrutura da base de dados é definida pelos clientes que usam a SDK do Firebase nas aplicações móveis. No caso desta dissertação, a estrutura é feita com classes Java na aplicação que definem os dados mencionados em cima. Quando ocorre uma escrita de um objeto através da SDK do Firebase, é enviado um JSON com a mesma estrutura da classe Java que substitui o valor atual no caminho especificado. A primeira chave do JSON é "users" e contém uma lista com os identificadores únicos de cada utilizador. Estes identificadores são gerados aleatoriamente pelo Firebase aquando da inserção do utilizador na base de dados. Depois, imediatamente a seguir, existem os campos "homeCharging", "vehicles" e "notifications" que representam respetivamente os dados do sistema de carregamento, os veículos adicionados e as notificações do utilizador.

```
{
  "users": {
    "userum": {
      "homeCharging":{},
      "vehicles":{},
      "notifications":{}
    },
    "userdois": {
      "homeCharging":{},
      "vehicles":{},
      "notifications":{}
    }
  }
}
```

Para o sistema de carregamento residencial, apenas se adiciona a corrente disponível para o carregamento, em amperes, e a data da sincronização.

```
{
  "homeCharging":{
    "current": 5,
    "lastSync":1501267946304
  }
}
```

O campo "lastSync" contém a data da sincronização em millisegundos desde 1 de janeiro de 1970, em Coordinated Universal Time (UTC).

Para as notificações, guardam-se dois booleanos, que representam o estado da notificação de carga completa e pouca carga, e identificadores únicos dos telemóveis registados, para que seja possível enviar as notificações. Estes identificadores são adicionados com valor de texto vazio porque não é possível adicionar chaves sem valor a um JSON. A alternativa seria usar um *array*, no entanto, neste caso, poderia haver colisões ao haver múltiplas escritas em simultâneo para a mesma posição. Como os identificadores são únicos, a escrita dos mesmos como uma chave no Firebase garante que não existe colisões.

```
{
  "notifications":{
    "lowNotification": true,
    "maxNotification": true,
    "tokens":{
      "cIjPwKPLXAoABASgteEtS...":"",
      "FbFDmuDOJyQ-TrvJCbdlB...":""
    }
  }
}
```

Para os veículos, guarda-se a matrícula, o nome (pode ser a marca ou modelo), dados dos sistemas de bateria e tração, localização e carregamento automático ou manual:

```
{
  "AA-22-55" : {
    "licensePlate" : "AA-22-55",
    "name" : "Tesla Series 3",
    "config" : {
      "manualCharge" : true,
      "smartCharge" : false
    },
    "location" : {
      "lastUpdate" : 1506965186716,
      "latitude" : 41.544886,
      "longitude" : -8.401586
    },
    "battery" : {
      "busVoltage" : 400,
      "charge" : 99,
      "charging" : true,
      "current" : 10,
      "lastSync" : 1508522283508,
      "gridCurrent" : 16,
      "gridVoltage" : 168,
      "temperature" : 30,
      "voltage" : 300
     },
    "engine" : {
      "controllerCurrent" : 50,
      "controllerPower" : 30,
      "controllerTemperature" : 60,
      "controllerVoltage" : 302,
      "lastSync" : 1508600612746,
      "temperature" : 50
    }
  }
}
```

De forma a autorizar apenas os próprios utilizadores a aceder aos dados dos seus veículos, definiram-se regras de segurança no Firebase. Estas regras de segurança aplicam-se tanto a leituras como a escritas dos dados. Apenas o próprio utilizador pode ler ou escrever dados dos seus veículos.

```
{
  "rules": {
    ".read": false,
    ".write": false,
    "users": {
      "$uid": {
        ".write": "$uid === auth.uid",
        ".read": "$uid === auth.uid"
      }
    },
  }
}
```

As propriedades de leitura e escrita na raiz definem a proibição de escrita de qualquer objeto. Isto evita casos em que atacantes tentem escrever dados de forma a apagar ou encher a base de dados. O identificador do utilizador autenticado é dado pela propriedade auth.uid. Caso o utilizador não esteja autenticado, esta propriedade é nula e não existe correspondência entre o identificador do utilizador e esse valor. Caso seja outro utilizador a tentar escrever ou ler os dados, também não existirá correspondência entre os auth.uid.

3.1.3 Funções

Foi desenvolvida apenas uma função para o Firebase. A função implementada envia notificações conforme o nível de bateria do veículo. O código foi adaptado de várias amostras em [30].

A função é invocada sempre que existe uma escrita no valor que corres-

ponde à carga da bateria, tal como pode ser visto na Figura 3.2.

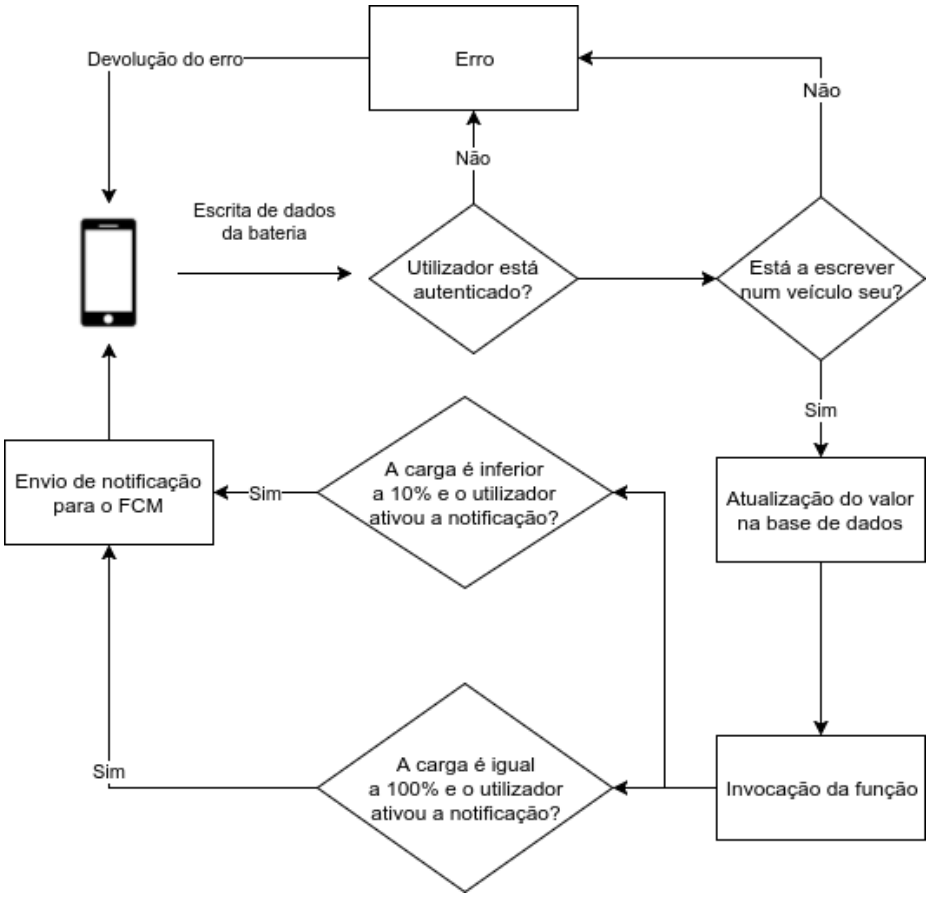


Figura 3.2: Fluxo de eventos quando ocorre uma mudança de carga da bateria.

Quando o nível de carga é inferior a 10%, é enviada uma notificação que alerta o utilizador sobre a baixa carga. Quando a carga atinge os 100%, também é enviada uma notificação ao utilizador. As notificações só são enviadas se o utilizador as ativar no perfil da aplicação.

Para inicializar o projeto de funções, é necessário primeiro instalar o NodeJS e o npm (gestor de pacotes). Depois de instaladas as dependências, instala-se a Firebase CLI (Command-line Interface) com o seguinte comando:

```
npm install -g firebase-tools
```

Depois da Firebase CLI estar instalada, executa-se os seguintes comandos para autenticar o utilizador e inicializar o projeto:

```
firebase login // Autentica o utilizador
firebase init functions // Inicializa o projeto
```

Depois do projeto estar inicializado, é criada uma pasta chamada "functions" que contém um ficheiro index.js onde podem ser adicionadas as funções. Para adicionar as funções ao Firebase, é necessário executar o seguinte comando na pasta do projeto:

```
firebase deploy
```

Depois deste comando, as funções podem ser vistas na consola do Firebase. Quando são executadas, é possível observar os logs para verificar se algo correu mal durante a execução. Também é possível observar o consumo atual de CPU e memória para que seja possível determinar se a função consome demasiados recursos e se precisa de ser otimizada.

Nas páginas seguintes é apresentado o código da função implementada em segmentos pequenos para que seja mais fácil de acompanhar. Para inicializar a SDK, usa-se o seguinte código antes da definição da função:

```
// Inicializar o SDK
const functions = require('firebase-functions');
const admin = require('firebase-admin');
admin.initializeApp(functions.config().firebase);
```

Depois, para obter notificações de escrita da carga da bateria, usa-se o seguinte código:

```
// Função notify
exports.notify = functions.database.ref('/users/{userUid}
    /vehicles/{vehicleUid}/battery/charge')
       .onWrite(event => {
         // Código executado quando ocorre uma escrita
       });
```

Os campos "userUid" e "vehicleUid" são usados para identificar qual é o veículo cuja bateria está a ser carregada ou descarregada. Para os obter, faz-se o seguinte:

```
// Obter identificadores
const userId = event.params.userUid;
const vehicleId = event.params.vehicleUid;
```

Para definir as notificações, define-se os objetos da seguinte forma:

```
// Payload da notificação de carga inferior a 10%
const payloadLow = {
  notification: {
    title: 'Bateria quase descarregada!',
    body: `A bateria do veículo ${vehicleId} está com menos de 10% de
        carga.`
  }
};

// Payload da notificação de carga máxima
const payloadMax = {
  notification: {
    title: 'Bateria carregada!',
    body: `A bateria do veículo ${vehicleId} está completamente carregada.`
  }
};
```

É necessário obter o valor de carga atual e o anterior, para que não sejam enviadas notificações repetidas caso a bateria continue a descer abaixo dos 10%:

```
// Valor da carga anterior
const previousValue = event.data.previous.val();

// Valor da carga atual
const newValue = event.data.val();
```

Depois é necessário determinar se o utilizador em questão ativou as notificações. Apenas são enviadas caso tenha escolhido no seu perfil.

```
const tokens = Object.keys(snapshot.child("tokens").val());
var maxNotificationEnabled = snapshot.child("maxNotification").val();
var lowNotificationEnabled = snapshot.child("lowNotification").val();
var payload = null;
if(original == 100 && previousValue != 100 && maxNotificationEnabled){
  payload = payloadMax;
}elseif(previousValue > 10 && original <= 10 && lowNotificationEnabled){
  payload = payloadLow;
}else{
  return;
}
return admin.messaging().sendToDevice(tokens,payload);
```

3.2 Configuração das Estações Periféricas

Nesta secção serão abordados todos os passos, desde o esquema de componentes ao código do *firmware* do controlador utilizado, que levaram à configuração das estações periféricas que ligam aos diferentes sensores (sistema de bateria, sistema de tração e parte residencial do sistema de carregamento). A localização das estações periféricas e da estação base podem ser vistas na Figura 3.3.

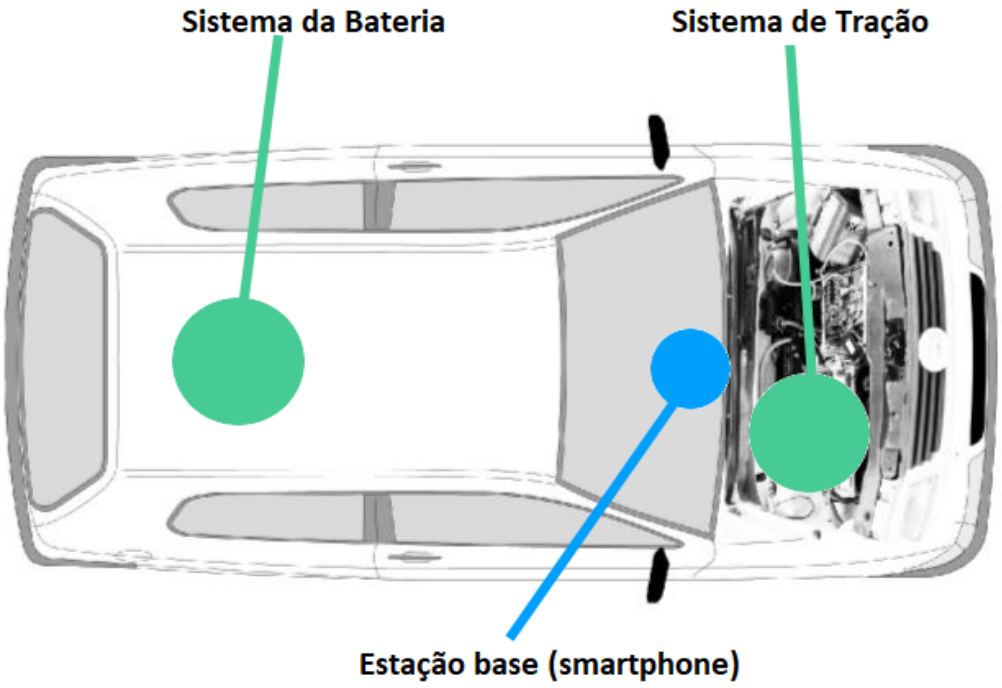


Figura 3.3: Localização das estações periféricas no veículo.

Foi utilizado um *kit* de desenvolvimento BLE da Cypress, o CY8CKIT-042-BLE-A, disponível em [31]. Este *kit* inclui uma placa de desenvolvimento (BLE Pioneer), dois módulos BLE e um adaptador Bluetooth USB para testes. O IDE utilizado para o desenvolvimento do código em C para este microcontrolador foi o PSoC Creator 4.1, disponível em [32].

3.2.1 Configuração dos Componentes

No âmbito do sistema desenvolvido nesta dissertação, todas as estações periféricas têm os seguintes componentes:

- BLE - componente que configura os parâmetros do serviço Bluetooth, tais como propriedades dos pacotes de anúncio, intervalos de ligação,

caraterísticas e notificações. Este componente é responsável também por gerir as ligações BLE e o envio de mensagens.

- Timer - componente que causa uma interrupção ao fim de um determinado período de tempo.
- Universal Asynchronous Receiver/Transmitter (UART) - componente que comunica com o sensor para a obtenção de dados.
- Bootloader - gere a escrita de código ou dados para a memória *flash* do dispositivo. Permite usar as portas RX e TX do componente UART via USB.

Estes componentes adicionam-se ao esquema a partir da lista de componentes disponível no lado direito do ecrã do PSoC Creator, como é apresentado na Figura 3.4.

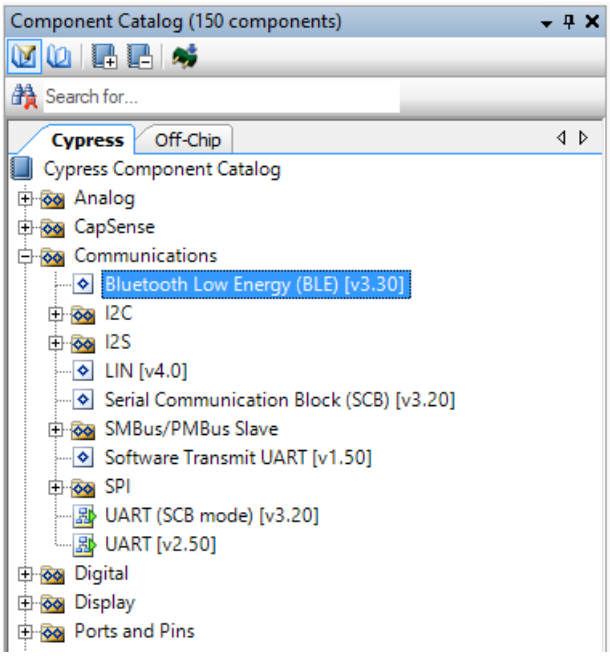


Figura 3.4: Catálogo de componentes disponíveis no PSoC Creator.

É possível pesquisar pelo componente pretendido ou procurar por categorias. Depois de encontrado o componente, arrasta-se para o *design* do projeto, que se encontra do lado esquerdo do catálogo, como pode ser visto na Figura 3.5.

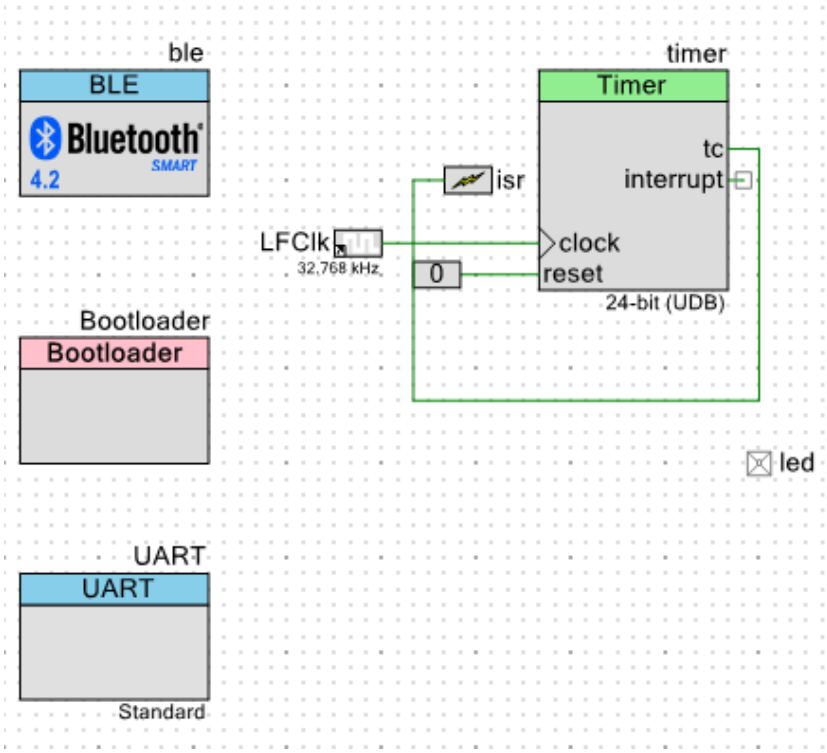


Figura 3.5: Componentes do PSoC Creator.

Neste ecrã é que se definem as ligações entre os componentes. Por exemplo, para o Timer, é necessário ligar o relógio, a interrupção e o valor 0 no reset.

No componente BLE, é necessário criar um serviço BLE e as caraterísticas de cada valor que esta estação enviará para a aplicação. As caraterísticas definem-se no separador "Profiles", tal como se pode ver na Figura 3.6.

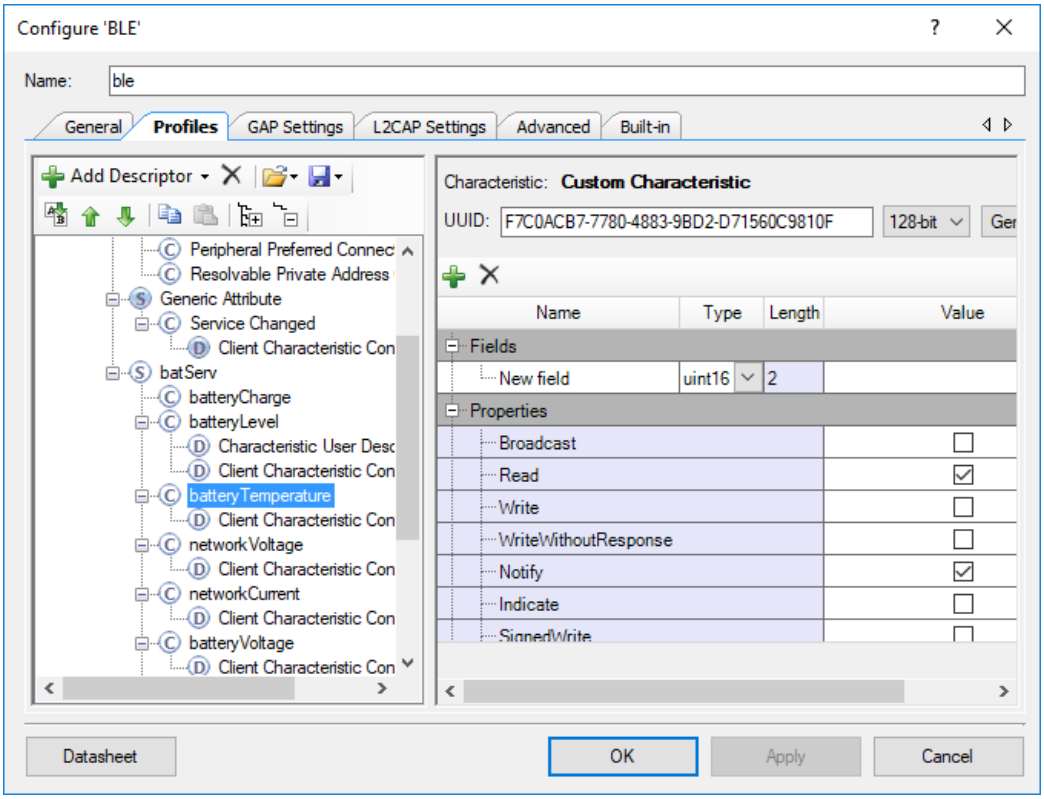


Figura 3.6: Propriedades da caraterística de temperatura da bateria.

Existe uma lista de caraterísticas pré-definidas que podem ser usadas. No entanto, caso estas não sejam apropriadas, podem-se criar outras recorrendo à opção "Custom Characteristic".

Todas as caraterísticas têm autorização de leitura e de notificações, à exceção da caraterística de corrente de carga do sistema de bateria, que permite mudar a corrente disponível para o carregamento. Ao selecionar a propriedade "notify", é criado automaticamente um descritor "Client Characteristic Configuration" com o UUID 00002902-0000-1000-8000-00805F9B34FB que permite ativar ou desativar as notificações remotamente. Este UUID é o mesmo para todas as caraterísticas.

Para a aplicação identificar a estação periférica, é necessário incluir o

UUID do serviço no pacote de anúncio. O pacote de anúncio identifica o serviço GATT do dispositivo Bluetooth. Desta forma, não é necessário estabelecer ligação a todos os dispositivos Bluetooth perto e fazer um pedido de lista dos serviços ou estabelecer ligação de forma manual. Esta definição altera-se no separador "GAP Settings", como se pode ver na Figura 3.7.

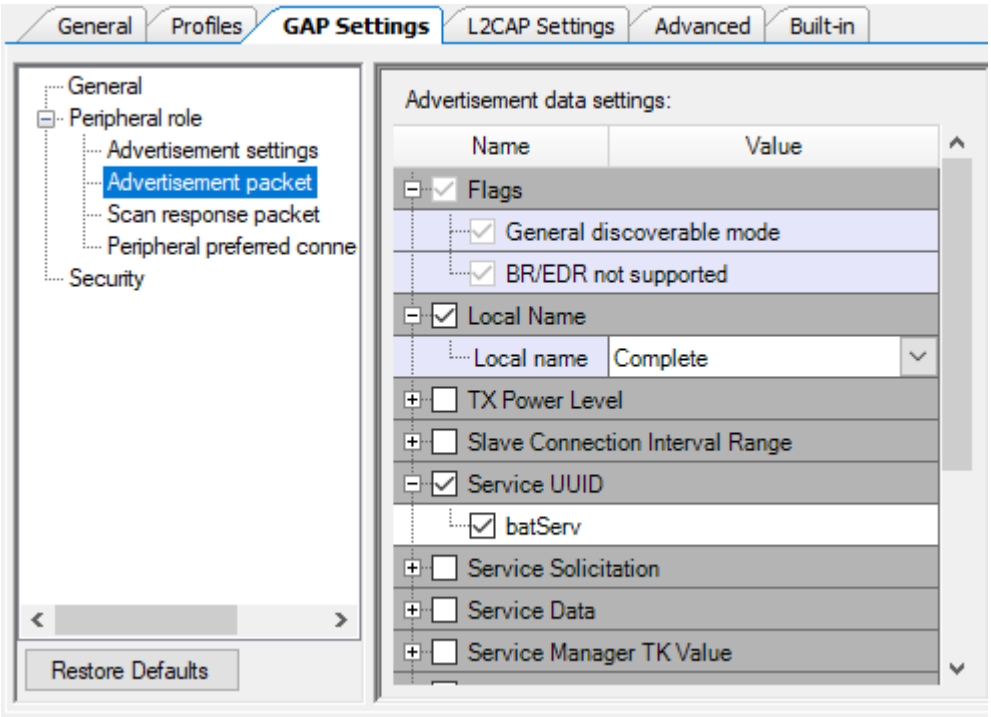


Figura 3.7: Alteração das definições do pacote de anúncio.

Na propriedade "Service UUID" escolhe-se o nome do serviço que foi criado anteriormente. Com esta definição, a aplicação Android apenas procura por dispositivos que incluem este serviço, o que permite estabelecer ligação de forma automática.

Para o Timer, é necessário configurar a frequência do relógio e a interrupção. O relógio escolhido foi o de baixa frequência (32.720 Hz), visto que a interrupção é invocada apenas a cada 500 milissegundos. Foi usado o projeto

em [33] como referência. A interrupção liga-se à saída TC (Terminal Count) e é invocada a cada período, definido na janela de configuração, tal como na Figura 3.8.

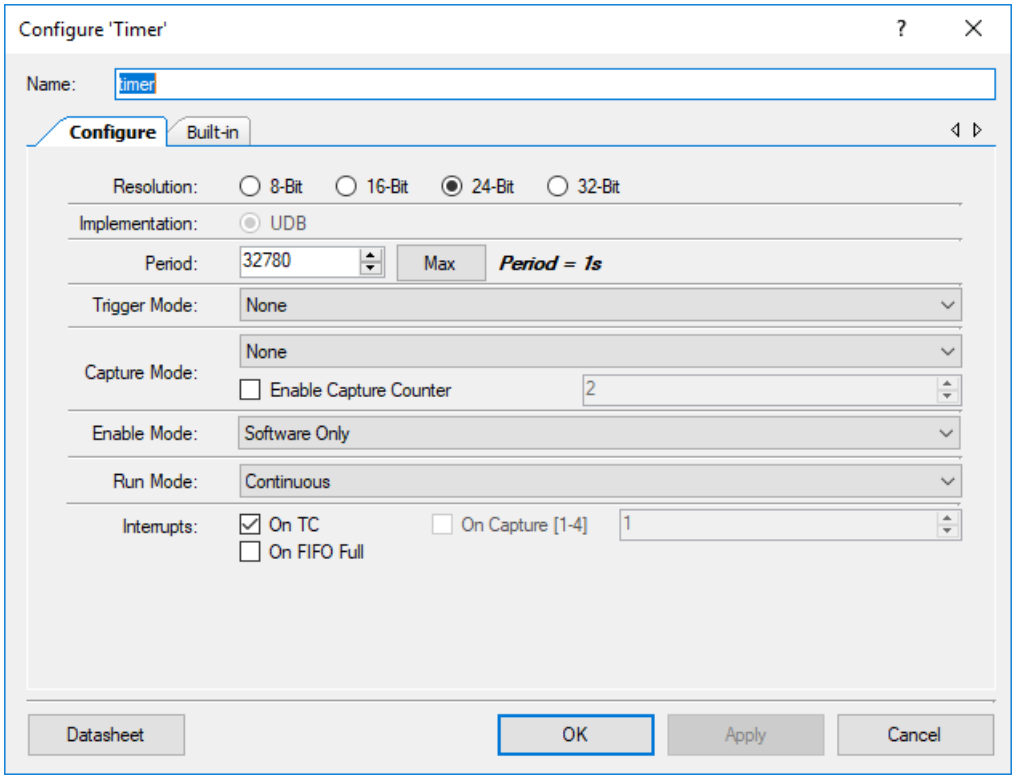


Figura 3.8: Configuração do componente Timer.

Para configurar a interrupção em código, é necessário definir uma função CY_ISR:

```
CY_ISR(isrFunction){
    // Código invocado a cada período
}
int main(){
    // Iniciar o Timer
    timer_Start();
    // Atribuir a função à interrupção
    isr_StartEx(isrFunction);
}
```

Para o componente UART, só foi preciso mudar o *baud rate* para o mesmo que o do Bootloader (Figura 3.9) e o tamanho dos *buffers*.

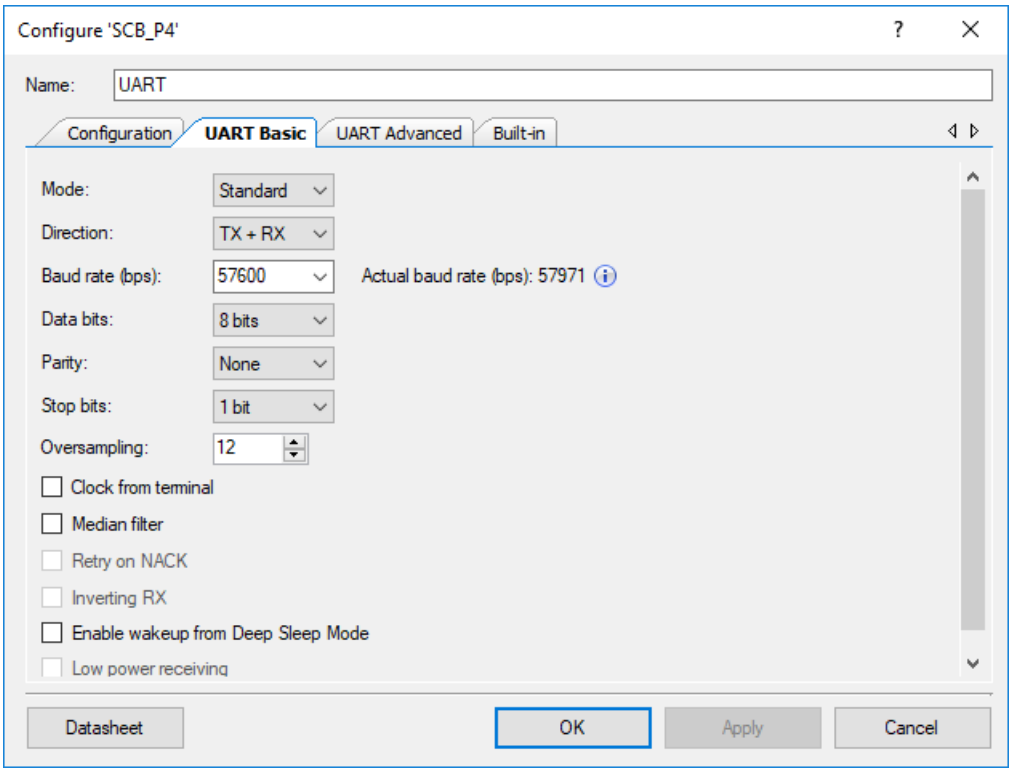


Figura 3.9: Configuração do componente UART.

Os *buffers* de RX e TX foram aumentados de 8 bytes para 32 bytes para que o pacote enviado pelo sensor seja recebido na totalidade ao mesmo tempo. Caso contrário, teria de ser desenvolvido um algoritmo de comunicação que lesse 8 bytes de cada vez e reconstruisse o pacote original.

3.2.2 Sincronização com os Sensores

A obtenção de dados dos sensores é feita via UART. O protocolo de comunicação usado para a ligação com os sistemas sensores definido na dissertação em [3] foi melhorado. Continua a funcionar por *polling*, com valor de 500 ms por defeito, mas configurável pela aplicação móvel. No entanto, nesta versão, é definido um pacote de *polling* de tamanho fixo com um cabeçalho mais pequeno, de 1 byte. O cabeçalho contém apenas o tipo de pedido, que pode ser de dados do sistema de bateria ou sistema de tração. Caso ocorra um erro do lado do sistema sensor, é enviado um pacote com um cabeçalho que indica que ocorreu um erro e outro byte que identifica a causa do erro. Caso contrário, é enviado um pacote com um cabeçalho que indica o sistema e os dados lidos. Os cabeçalhos válidos são os seguintes:

- 0x01 - Pedido de dados do sistema da bateria.
- 0x02 - Pedido de dados do sistema de tração.
- 0x03 - Pedido de dados do sistema de carregamento.
- 0x04 - Mudança de valor da corrente de carregamento.
- 0xFF - Erro.

Os sistemas sensores devolvem um pacote que contém o mesmo cabeçalho do pedido de *polling*, ou um cabeçalho de erro caso algo tenha corrido mal. Na Figura 3.10 está representado o conteúdo do pacote de dados que chega do sensor do sistema da bateria.

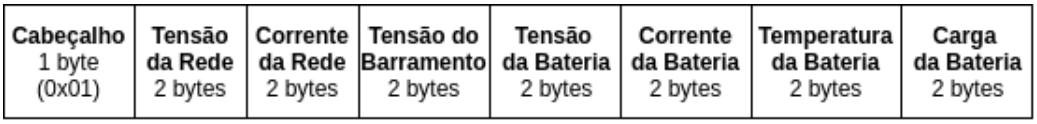


Figura 3.10: Conteúdo do pacote de dados do sistema da bateria.

Os valores de dois bytes suportam amostras de 16 bits, mas são utilizados apenas 12 bits atualmente. Estes valores representam o valor amostrado e não têm unidade. São usados para o cálculo de valores reais com casas decimais. Como este protocolo ainda não foi implementado do lado dos sistemas sensores, foi desenvolvido um programa em Java que emula os valores, fazendo-os variar ao longo do tempo. Este programa é executado num PC e faz o papel do sistema de bateria. Foi utilizada a biblioteca jSSC (Java Simple Serial Connector) disponível em [34] para facilitar a comunicação com a porta série do microcontrolador. Os valores são gerados a uma frequência constante e fazem-se variar entre os mínimos e máximos, que podem ser vistos nas Tabela 3.1 e 3.2.

Tabela 3.1: Tabela de valores do sistema de bateria [3].

Parâmetro	Mínimo	Máximo
Tensão da rede	0 V	240 V
Tensão do barramento	0 V	20 A
Tensão da bateria	0 V	360 V
Corrente da bateria	0 A	13 A
Temperatura da bateria	0 ºC	100 ºC
Carga da bateria	0 %	100 %

Tabela 3.2: Tabela de valores do sistema de tração [3].

Parâmetro	Mínimo	Máximo
Tensão do controlador	0 V	400 V
Corrente do controlador	0 V	30 A
Potência do controlador	0 V	600 kW
Temperatura do motor	0 °C	13 °C
Temperatura do controlador	0 °C	100 °C

O valor inteiro ($V_{adquirido}$) gerado pelo programa Java depende de uma variável que é incrementada a cada 200 ms. Esta variável representa o ângulo na função cosseno que depois se multiplica pelo valor máximo de cada parâmetro.

```
private float generateRawData(int max, int sync) {
    return (float) (Math.abs(Math.cos(Math.toRadians(sync))) * max);
}
```

Depois de gerado o valor real, é necessário aplicar a seguinte fórmula para obter o valor adquirido:

$$V_{adquirido} = \frac{V_{real} \times (2^R - 1)}{V_{max} - V_{min}}$$

R representa a resolução do ADC, que neste caso é de 12 bits. Este valor medido depois é convertido para bytes e colocado na posição correta do pacote de dados. A conversão para o valor real é feita depois na aplicação, aplicando a fórmula:

$$V_{real} = \frac{V_{max} - V_{min}}{2^R - 1} \times V_{adquirido}$$

3.2.3 Sincronização com a Aplicação

O envio de dados para a aplicação é feita à base de notificações. Para isso, são guardados os valores recebidos pelos sensores num *array* de duas posições em que a primeira posição contém o valor atual e a segunda o valor antigo. Quando o valor atual é diferente do valor antigo, é enviada uma notificação para a aplicação. O fluxo de eventos pode ser visto na Figura 3.11.

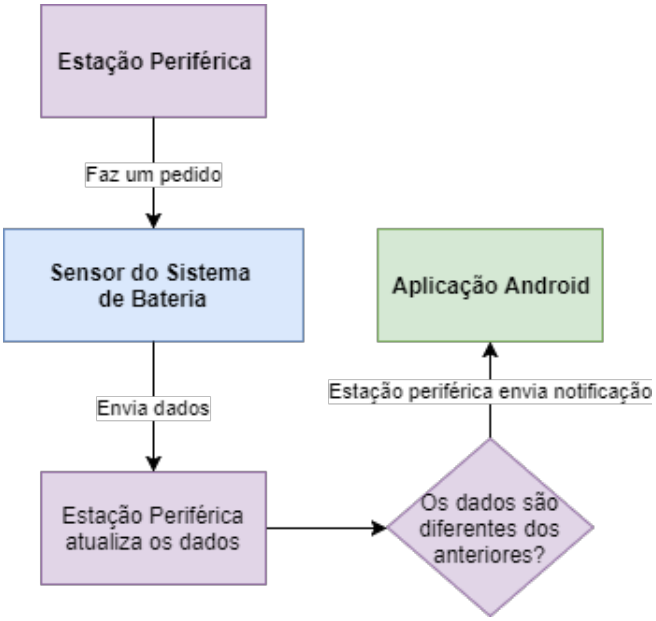


Figura 3.11: Fluxo de eventos de uma estação periférica.

A aplicação Android envia um parâmetro para todas as estações periféricas: a taxa de refrescamento dos dados do sensor. Quando um novo valor da taxa de refrescamento é recebido, o período do Timer é alterado da seguinte forma:

```
int updateRefreshPeriod(int ms){
  int period = TIMER_CLOCK_PERIOD * ms / 1000;
  timer_WritePeriod(period);
}
```

Caso a aplicação Android esteja ligada à estação periférica do sistema de carregamento, é enviado também um parâmetro que representa a corrente disponível para o carregamento. Depois da estação periférica receber esse novo valor, envia para o sistema de carregamento.

3.3 Desenvolvimento do Programa para o Raspberry Pi

Foi desenvolvido um programa para o Raspberry Pi 3 que lê dados de um sensor emulado com o *kit* BLE configurado na secção anterior. Este programa foi desenvolvido em Java e usa a biblioteca TinyB [35] para a comunicação Bluetooth e a biblioteca Firebase Admin [36] para a sincronização dos dados para o Firebase. O Raspberry Pi foi configurado para incluir o sistema operativo Raspbian, uma distribuição Linux baseada no Debian (versão Jessie). O programa foi desenvolvido usando o IDE IntelliJ IDEA, versão 2017.

3.3.1 Controlo da Corrente de Carregamento

A corrente de carregamento é calculada com base na corrente lida a partir de um sensor, representada como I_{casa} nas equações seguintes. Para evitar o disparo do disjuntor da residência, é necessário garantir que a corrente de carregamento somada com a corrente usada pela habitação seja inferior à corrente máxima disponível:

$$I_{carregamento} + I_{casa} < I_{max}$$

Como o valor da corrente de carregamento é comunicado para o sistema de carregamento do veículo através da Internet, existe um atraso da comu-

nicação. Por causa disso, definiu-se um valor que limita a corrente de carregamento máxima a uma percentagem da corrente máxima, representado como "maxThreshold":

$$I_{carregamento} = I_{max} \times maxThreshold - I_{casa}$$

No programa, este valor foi definido como 0,85 por defeito.

3.3.2 Instalação das Dependências

TinyB

A escolha desta biblioteca deveu-se principalmente à facilidade de uso da API e à documentação existente na Internet. Para usar a biblioteca, é necessário compilá-la com o CMake 3.1 ou superior e o BlueZ 5.37 ou superior, visto que o suporte ao perfil GATT não existe em versões anteriores. Foi usado o tutorial indicado em [37] para a configuração das dependências necessárias para a compilação da biblioteca.

Depois de compilar a biblioteca, foi gerado um ficheiro tinyb.jar para ser importado no programa. Como projeto criado usa o Gradle, basta copiar o ficheiro jar para a pasta libs, graças à seguinte configuração no ficheiro build.gradle:

```
dependencies {
    compile fileTree(dir: 'libs', include: '*.jar')
}
```

Não é necessário carregar qualquer biblioteca nativa, visto que o próprio TinyB já o faz durante o arranque do programa graças ao seguinte código na classe BluetoothManager:

```
static {
  try {
    System.loadLibrary("javatinyb");
```

```
  } catch (UnsatisfiedLinkError var1) {
    System.err.println(''Native code library failed to load.\n'' + var1);
    System.exit(-1);
  }
}
```

Caso a instalação não tenha sido bem sucedida e as bibliotecas nativas não tenham sido copiadas para as pastas corretas, o programa irá emitir a exceção UnsatisfiedLinkError e terminará de imediato.

Firebase Admin SDK

Para adicionar a biblioteca Firebase Admin, é necessário adicionar a seguinte linha no ficheiro build.gradle:

```
dependencies {
    compile 'com.google.firebase:firebase-admin:5.4.0'
}
```

A versão 5.2.0 era a mais recente na altura em que a dissertação foi escrita.

Para inicializar esta biblioteca, é necessário obter um ficheiro de configuração na consola do Firebase, indo a "Definições" e ao separador "Contas de Serviço". O ficheiro contém uma chave privada necessária para a autenticação do programa e outros valores que identificam unicamente a conta do Firebase.

Depois de o ficheiro estar no Raspberry Pi, é necessário inicializar a biblioteca da seguinte forma:

```
FileInputStream serviceAccount =
  new FileInputStream("/caminho/config.json");

FirebaseOptions options = new FirebaseOptions.Builder()
  .setCredential(FirebaseCredentials
  .fromCertificate(serviceAccount))
  .setDatabaseUrl("https://cepiumapp.firebaseio.com")
  .build();
```

```
FirebaseApp.initializeApp(options);
```

Só depois destes passos é que é possível escrever com sucesso na base de dados do Firebase.

3.3.3 Sincronização dos Dados

O programa desenvolvido para o Raspberry Pi usa o mecanismo de notificações do BLE para a obtenção dos dados de corrente. O ciclo de vida do programa é o seguinte:

1. Descoberta dos dispositivos BLE que fornecem o serviço de corrente.
2. Estabelecer ligação ao dispositivo encontrado.
3. Subscrever as notificações de alteração do valor de corrente.
4. Escrever na base de dados do Firebase.

Na Figura 3.12 podemos ver um fluxograma que descreve o comportamento do programa.

A descoberta do sensor é feita através da classe BluetoothManager:

```
BluetoothManager manager = BluetoothManager.getBluetoothManager();
manager.startDiscovery();
```

Para este método ser bem sucedido, é necessário o adaptador Bluetooth estar ligado. Caso esteja desligado, é possível ligá-lo com o programa hciconfig do pacote bluez. Para isso, temos de obter a lista de adaptadores com:

```
hciconfig
```

Na Figura 3.13 podemos ver o resultado da execução do programa e o nome do adaptador (hci0).

Para o ligar, basta executar:

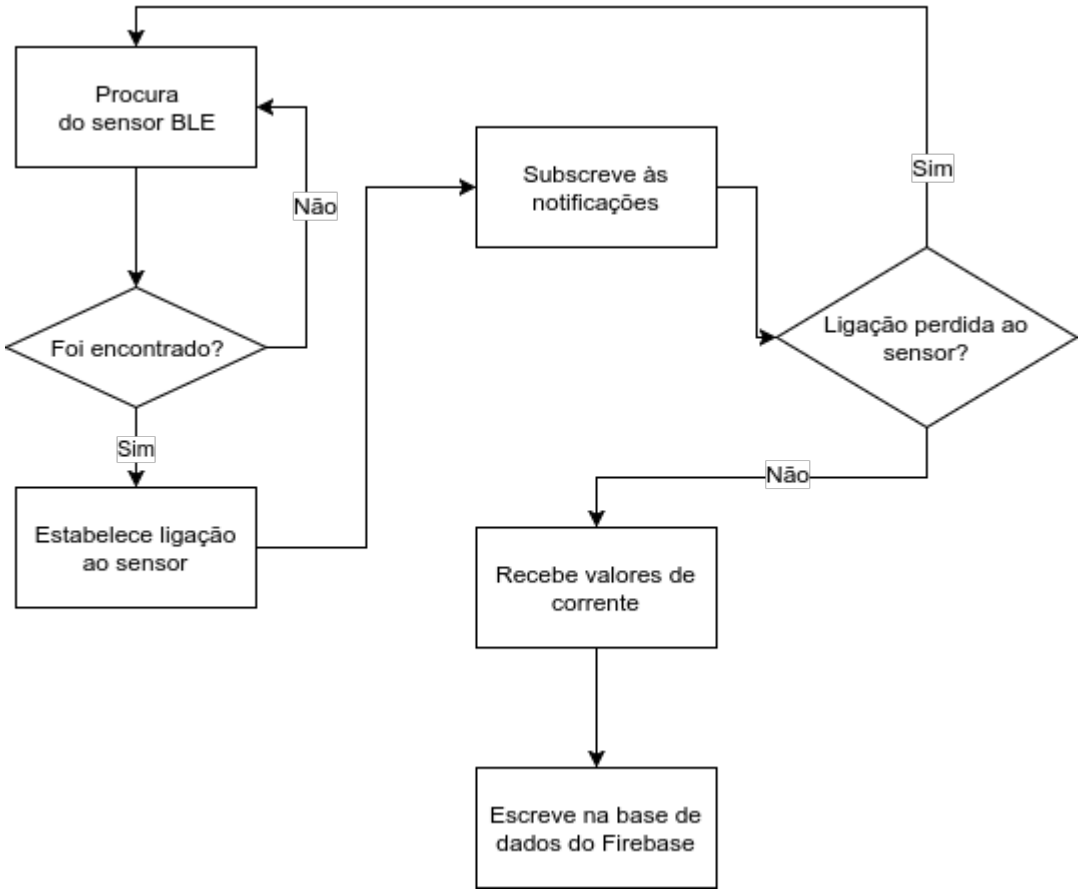


Figura 3.12: Fluxograma do programa do Raspberry Pi.

```
ruben@rubenPC:~$ hciconfig
hci0:   Type: Primary  Bus: USB
        BD Address: 00:1A:7D:DA:71:13  ACL MTU: 310:10  SCO MTU: 64:8
        DOWN
        RX bytes:574 acl:0 sco:0 events:30 errors:0
        TX bytes:368 acl:0 sco:0 commands:30 errors:0
```

Figura 3.13: Resultado da execução do comando "hciconfig"

```
sudo hciconfig hci0 up
```

Depois de ativada a descoberta dos dispositivos BLE, é necessário ligar ao dispositivo que fornece o serviço de corrente que queremos. Para isso,

é necessário iterar sobre a lista de dispositivos armazenados em memória pelo BluetoothManager e verificar se contém o serviço com o UUID que corresponde ao nosso serviço de corrente fornecido pelo sensor.

```
List<BluetoothDevice> devices = bluetoothManager.getDevices();

for (BluetoothDevice device : devices) {
    String[] uuids = device.getUUIDs();
    for (String uuid : uuids) {
        if (uuid.equals(UUID_SERVICE)) {
            // Dispositivo encontrado
            return device;
        }
    }
}
```

Depois de encontrado o dispositivo, é necessário invocar o método connect() para estabelecer uma ligação. Só depois da ligação estar estabelecida é que é possível subscrever às notificações. Para isso, é necessário obter a BluetoothGattCharacteristic que corresponde à caraterística definida no sensor. Esta caraterística obtém-se a partir da classe BluetoothGattService, que expõe o método getCharacteristics():

```
for (BluetoothGattService s: services) {
    if (s.getUUID().equals(UUID_SERVICE)) {
        List<BluetoothGattCharacteristic> cs = service.getCharacteristics();
        for (BluetoothGattCharacteristic c : cs) {
            if (c.getUUID().equals(UUID_CHARACTERISTIC)) {
          // Carateristica encontrada
                return characteristic;
            }
        }
    }
}
```

Depois de obtida a caraterística, basta subscrever às notificações através do método enableValueNotifications. Este método recebe como parâmetro uma interface BluetoothNotification genérica que serve para comunicar o

valor de corrente à classe responsável pela escrita no Firebase, a FirebaseStorage. Esta classe tem um metódo setCurrent que escreve o valor de corrente no Firebase, juntamente com a data da leitura:

```
public void setCurrent(float current, long lastUpdate) {
  FirebaseDatabase db = FirebaseDatabase.getInstance();
    Map<String, Object> data = new HashMap<>();
    data.put("current", current);
    data.put("lastSync", lastUpdate);
    db.getReference("homecharging").updateChildren(data);
}
```

3.4 Desenvolvimento da Aplicação Móvel

Nesta secção é abordado o desenvolvimento da aplicação móvel que comunica com os sensores no veículo e com o Firebase. São dados excertos de código relevante para a execução de certas tarefas, tais como a escrita de dados no Firebase e a leitura de dados via BLE.

3.4.1 Instalação das Dependências

A aplicação requer algumas dependências para a integração com os serviços do Google Play, com o Firebase e para a facilidade de desenvolvimento. As dependências adicionam-se no ficheiro build.gradle gerado pelo Android Studio quando se cria a aplicação. As dependências do Firebase são as seguintes:

```
// Base de dados em tempo real
'com.google.firebase:firebase-database:11.4.0'
// Firebase Cloud Messaging
'com.google.firebase:firebase-messaging:11.4.0'
// Autenticação
'com.google.firebase:firebase-auth:11.4.0"
```

As versões acima eram as mais recentes à data de escrita da dissertação. Para os serviços do Google Play, adiciona-se as seguintes dependências no

build.gradle:

```
// Login da Google
'com.google.android.gms:play-services-auth:11.4.0'
// API do Google Maps
'com.google.android.gms:play-services-maps:11.4.0'
// Serviços de localização do Google Play
'com.google.android.gms:play-services-location:11.4.0'
```

Depois adiciona-se o *plugin* dos serviços do Google Play no build.gradle do projeto raiz:

```
buildscript {
    repositories {
        jcenter()
        google()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.0.0'
        classpath 'com.google.gms:google-services:3.1.1'
    }
}
```

Este *plugin* ativa-se na última linha do ficheiro build.gradle da aplicação:

```
apply plugin: 'com.google.gms.google-services'
```

É também necessário colocar o ficheiro google-services.json (que é gerado pela consola do Firebase) na pasta do módulo da aplicação. Este ficheiro contém informação sobre o projeto do Firebase, tais como nome e identificador da aplicação e também o endereço *web* para acesso à base de dados.

Por fim, é necessário ativar a API do Google Maps. Para isso, cria-se uma chave na consola de APIs da Google e adiciona-se o nome do pacote da aplicação e a impressão digital SHA1, tal como no Firebase. Depois de criada a chave, adiciona-se no AndroidManifest.xml:

```
<meta-data
  android:name="com.google.android.geo.API_KEY"
  android:value="@string/google_maps_key" />
```

3.4.2 Arquitetura da Aplicação

A aplicação usa a arquitetura MVP para a camada de apresentação e uma divisão de pacotes mista entre funcionalidades e camadas. No primeiro nível dos pacotes, a divisão é feita por camada (camada de dados, interface gráfica e serviços). No segundo nível, é feita por funcionalidade: informações de bateria, localização e perfil, por exemplo. Na Figura 3.14 podemos ver a lista de pacotes raiz da aplicação.

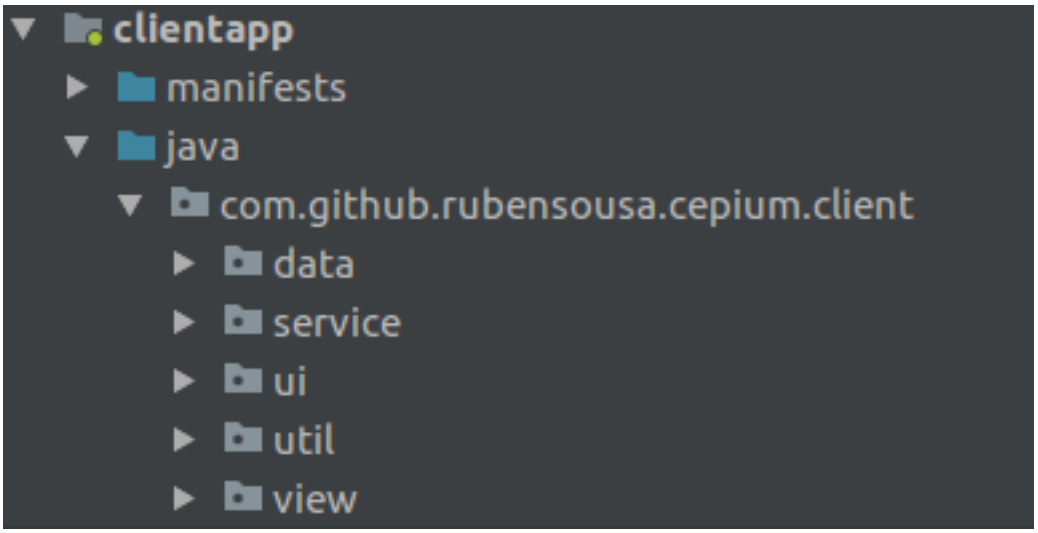


Figura 3.14: Pacotes da aplicação cliente.

A função de cada pacote é a seguinte:

- data - contém Plain Old Java Objects (POJOs) que representam os dados da aplicação (bateria, sistema de tração, localização) e repositórios para aceder aos mesmos dados.
- service - contém o serviço de Bluetooth e classes responsáveis por gerir as ligações e a obtenção de dados.

- ui - contém classes responsáveis pela interface gráfica (Activities e Fragments).
- util - contém classes utilitárias que ajudam a reduzir a repetição de código noutras classes.
- view - contém Views customizadas.

Dentro do pacote "ui" encontram-se outros pacotes que definem as funcionalidades da aplicação. Os pacotes são os seguintes:

- login - ecrã de autenticação.
- profile - ecrã do perfil do utilizador. Contém as preferências de notificações e o botão de sair.
- battery - ecrã de informações da bateria.
- engine - ecrã de informações do sistema de tração.
- location - ecrã que contém a localização do veículo num mapa do Google Maps.
- vehicles - ecrã que contém a lista de veículos do utilizador e onde este pode adicionar um.

Cada um destes pacotes inclui a View e o Presenter do padrão MVP e outras classes secundárias responsáveis pela interface gráfica da aplicação. A View e o Presenter são definidos por uma interface. Por exemplo, para o ecrã de informações da bateria, temos a seguinte interface:

```
public interface BatteryContract {
    interface View extends CarContract.View<Presenter> {
        void updateManualCharge(boolean enable);
        void updateSmartCharging(boolean enable);
```

```
        void updateData(BatteryData data);
    }
    interface Presenter extends CarContract.Presenter<View> {
        void enableCharge(boolean enable);
        void enableSmartCharge(boolean enable);
    }
}
```

Ao definir esta interface, sabemos quais são as funcionalidades a implementar para este ecrã. Neste caso, é criado um BatteryFragment que implementa a interface View contida no BatteryContract e um BatteryPresenter que implementa o Presenter. Cada Presenter recebe um repositório de acordo com os dados que necessita. Por exemplo, para o BatteryPresenter, é necessário um BatteryRepository. Cada repositório tem acesso a uma cache e à API do Firebase para a obtenção dos dados, como pode ser visto na Figura 3.15.

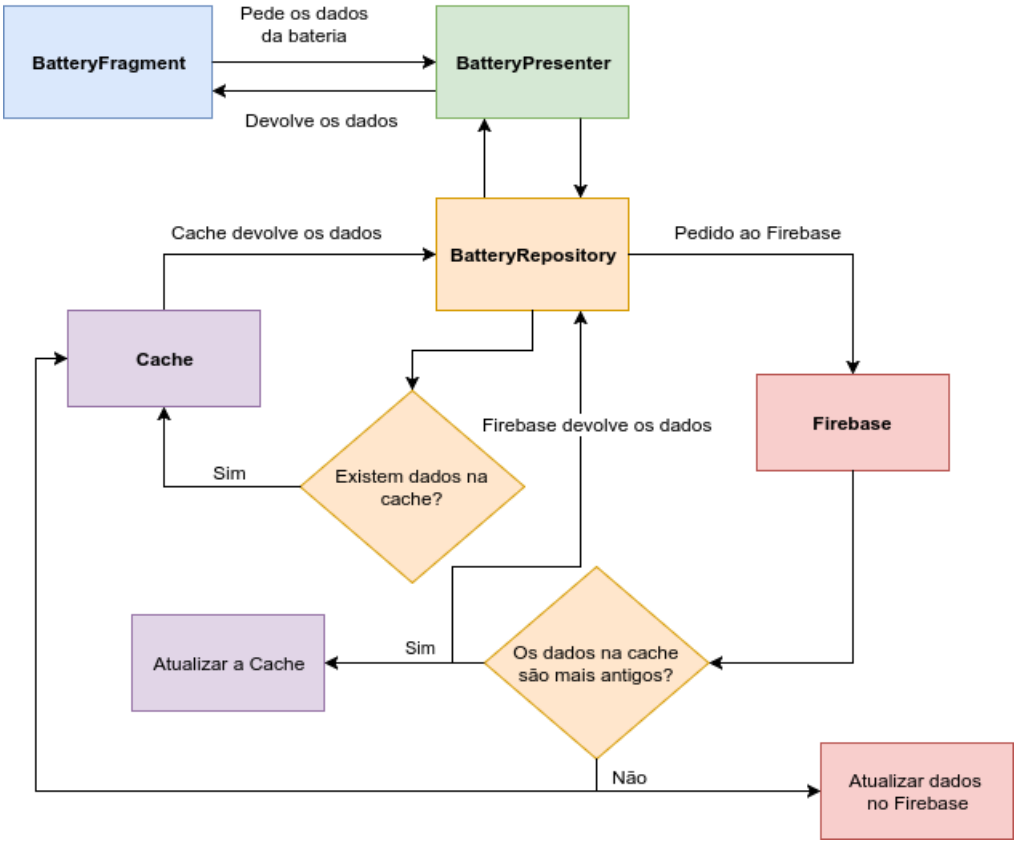


Figura 3.15: Fluxo de eventos no ecrã de informações da bateria.

Sempre que é feito um pedido de dados, é feito um pedido ao Firebase e à cache em simultâneo. Caso exista dados na cache, estes são devolvidos imediatamente. Caso contrário, aparece um indicador de progresso no ecrã enquanto o Firebase não devolve o resultado do pedido. Quando são obtidos os dados do Firebase, é feita uma comparação com o conteúdo da cache. Caso a cache seja antiga, é atualizada com o resultado do Firebase e caso o Firebase tenha os dados desatualizados, é enviado o conteúdo da cache. Com isto, a aplicação funciona enquanto não está ligada a Internet e é sempre possível verificar as últimas informações sincronizadas.

3.4.3 Autenticação

Tal como já foi referido em secções anteriores, a autenticação na aplicação é gerida pelo Firebase. O método de autenticação escolhido foi o da conta da Google, visto que todos os utilizadores precisam de uma para conseguirem instalar aplicações a partir do Google Play.

Quando a aplicação é iniciada, verifica-se se o utilizador está autenticado antes de mostrar o conteúdo. Isso faz-se da seguinte forma:

```
FirebaseAuth auth = FirebaseAuth.getInstance();
if (auth.getCurrentUser() == null) {
    // Utilizador não autenticado
}else{
  // Utilizador autenticado
}
```

Caso o utilizador não esteja autenticado, é redirecionado para o ecrã de autenticação (LoginActivity) que pode ser visto na Figura 3.16. A verificação do estado é feita no método onCreate da MainActivity.

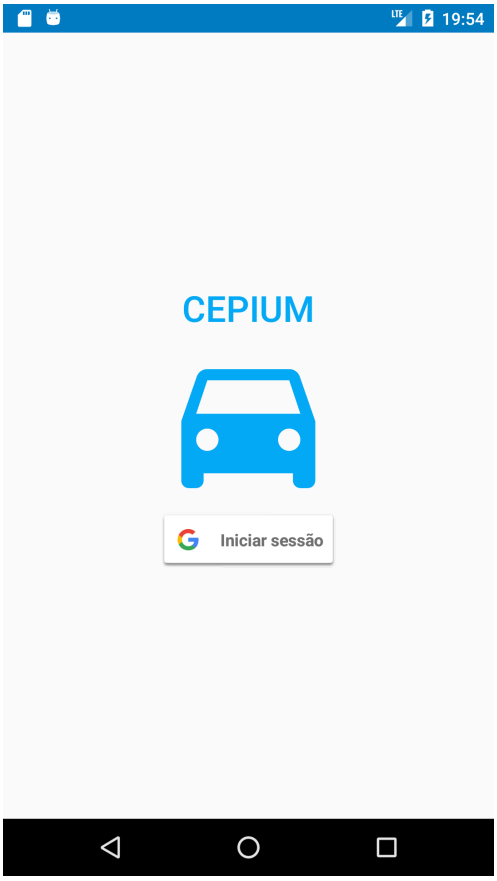


Figura 3.16: Ecrã de autenticação da aplicação.

Quando o utilizador clica no botão, é lançada uma Activity onde é possível escolher a conta da Google. Depois de escolhida a conta, a LoginActivity recebe o resultado no método onActivityResult. Nesse método, obtem-se a conta da Google (GoogleSignInAccount). Depois usa-se a API do Firebase para autenticar usando um *access token* que está nessa classe.

```
private void login(GoogleSignInAccount account) {
    FirebaseAuth auth = FirebaseAuth.getInstance();
    AuthCredential c = GoogleAuthProvider.getCredential(
    account.getIdToken(), null);
    auth.signInWithCredential(c).addOnCompleteListener(this, this);
}
```

3.4.4 Lista de Veículos

Quando o utilizador entra na aplicação pela primeira vez, é-lhe pedido para adicionar pelo menos um veículo, tal como pode ser visto na Figura 3.17. Esta verificação é feita descarregando a chave ”vehicles”e verificando o número de valores.

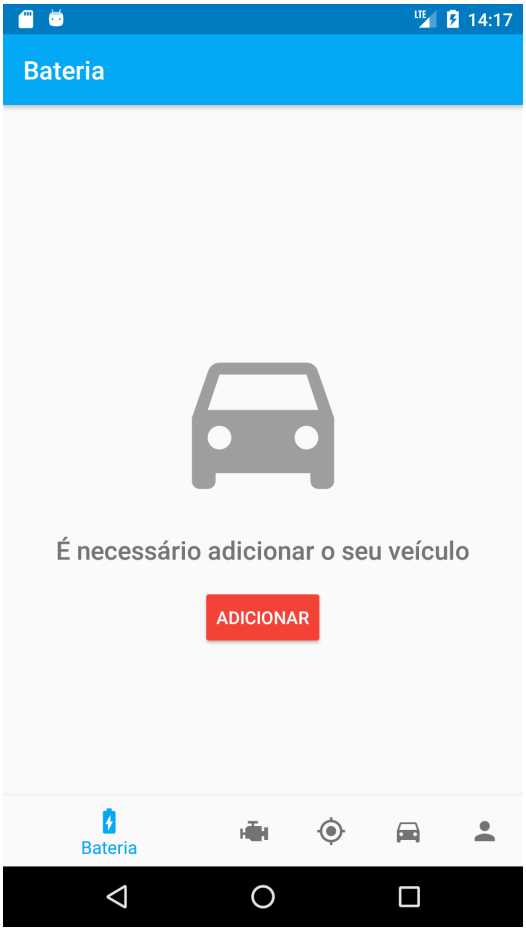


Figura 3.17: Ecrã de bateria quando não existem veículos.

Ao clicar no botão “Adicionar”, o utilizador é redirecionado para o ecrã de veículos (3.18). Neste ecrã existe um botão que abre uma Activity onde o utilizador define o nome e a matrícula do veículo.

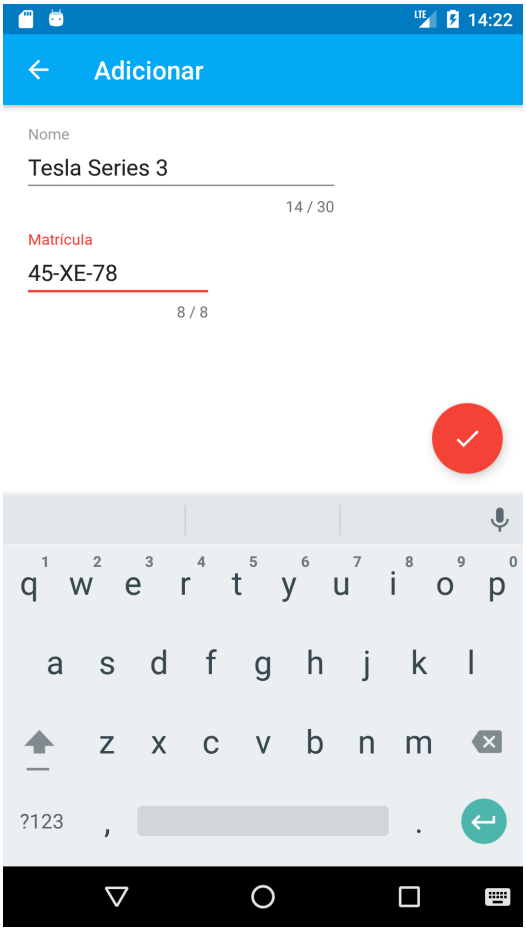


Figura 3.18: Ecrã onde se adiciona um veículo.

Depois de adicionado o nome e a matrícula, o veículo é adicionado ao Firebase e o utilizador regressa ao ecrã anterior. No entanto, agora aparece o veículo que foi adicionado, como pode ser visto na Figura 3.19.

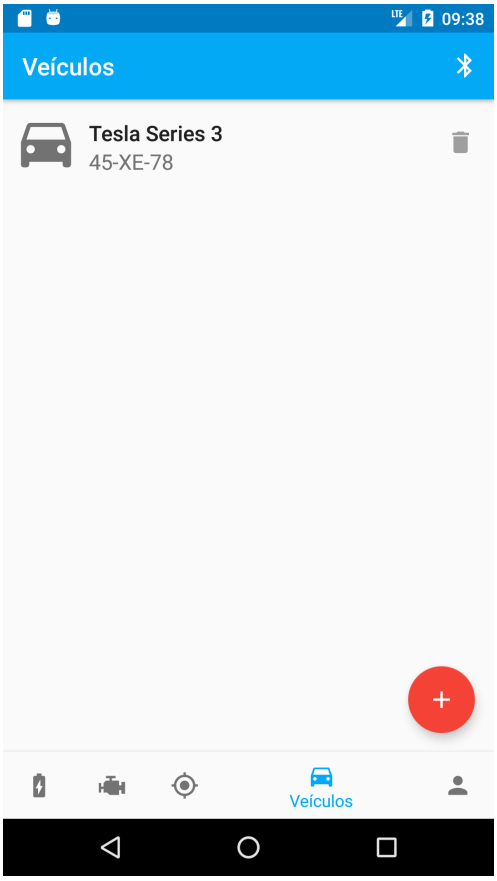


Figura 3.19: Lista de veículos do utilizador.

3.4.5 Sincronização com o Veículo

Para iniciar a sincronização com o veículo, é necessário clicar no ícone de Bluetooth que se encontra na Toolbar. Ao clicar nesse ícone, é iniciado o serviço de Bluetooth (BleService) e aparecerá uma notificação como na Figura 3.20.

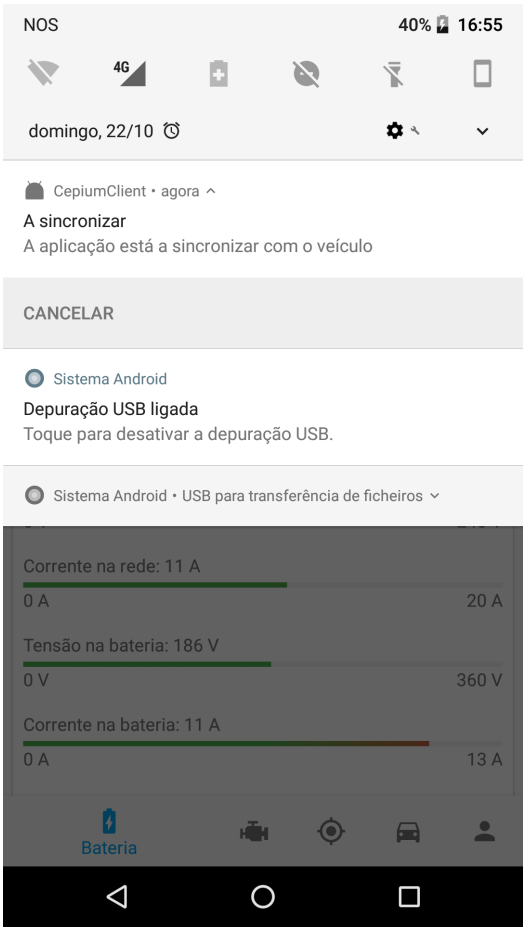


Figura 3.20: Notificação de sincronização com o veículo.

O serviço de Bluetooth inicia uma Thread para fazer a procura de dispositivos que implementem os serviços definidos na configuração do *kit* BLE. A procura destes dispositivos é feita pela classe BleScanner. O utilizador não tem de escolher o dispositivo ao qual se quer ligar. Quando um dispositivo é encontrado, o BleScanner passa-o ao BleConnectionManager para estabelecer uma ligação. Depois da ligação estar estabelecida, o BleConnectionManager aguarda pela resolução do serviço e pela lista de caraterísticas do mesmo. Assim que a lista de caraterísticas for obtida, são pedidas notificações para

cada uma. Este processo pode ser visto na Figura 3.21.

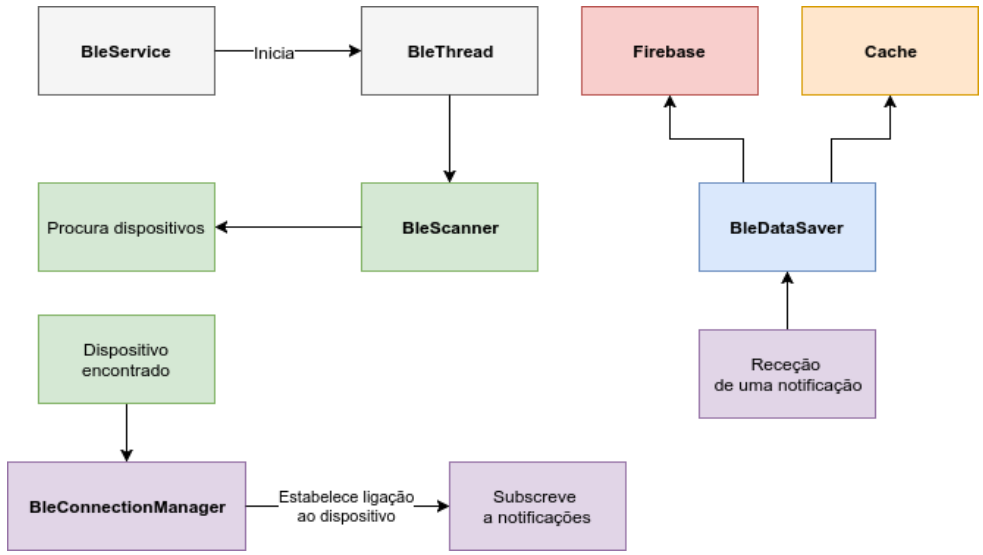


Figura 3.21: Fluxo de eventos do serviço de Bluetooth.

Quando uma notificação é recebida, o BleDataSaver interpreta os dados da caraterística cujo valor mudou e guarda-os tanto na cache local como no Firebase.

```
@Override
public void onCharacteristicChanged(BluetoothGatt gatt,
        BluetoothGattCharacteristic gattChar) {
    super.onCharacteristicChanged(gatt, gattChar);
    dataSaver.saveData(gattChar);
}
```

A interpretação dos dados é feita consoante o tipo dos dados que se espera para uma dada caraterística. Por exemplo, para a carga, sabemos que é representada por um inteiro de 0 a 100. Como é recebido um inteiro de 16 bits que representa o valor medido, é necessário aplicar a seguinte fórmula:

$$V_{real} = \frac{V_{max} - V_{min}}{2^R - 1} \times V_{medido}$$

V representa um valor sem unidade e não um valor de tensão. Este valor depois é atribuído à variável correta, depois de feita a comparação do Universally Unique Identifier (UUID). Por exemplo, caso chegue 4095 no valor de carga, o valor real é de:

$V_{real} = \frac{100-0}{2^{12}-1} \times 4095 = 100\%$

Como a estação periférica envia um inteiro de 16 bits sem sinal, chega um *array* de bytes com duas posições. Para interpretar esses bytes como um inteiro, é feito o seguinte:

```
private int parseUnsignedInt16(byte[] bytes) {
    if (bytes.length != 2) {
        return 0;
    }
    return bytes[0] & 0xFF + ((bytes[1] & 0xFF) << 8);
}
```

Primeiro aplica-se uma máscara a ambos os bytes, de forma a converter um número com sinal num sem sinal. Depois, soma-se o primeiro número ao segundo número depois de este último ser deslocado 8 bits para a esquerda.

Visto que pode haver receção de pacotes a cada meio segundo, a taxa de atualização ao Firebase é costumizável para evitar um consumo de dados excessivo. Cada repositório define a sua taxa de atualização. Quando os dados são gravados na cache, é verificado a data da última sincronização:

```
public void save(T data) {
    cachedValue = data;
    cache(data);
    long time = System.currentTimeMillis();
    if (time - lastSync > getSyncFrequency()) {
        upload(data);
        lastSync = time;
    }
}
```

A classe BleDataWriter é a responsável por mudar o estado de carregamento do veículo e a taxa de atualização de obtenção dos dados dos sensores.

Para isso, é feita uma subscrição de dados ao ChargerRepository, o repositório que devolve o estado do carregador e ao ProfileRepository. Quando existe uma alteração no valor da variável de carregamento manual, o BleDataWriter muda o valor de uma caraterística BLE do sensor da bateria:

```
@Override
public void onDataLoaded(ChargerData data) {
    int value = data.isManualCharge() ? 1 : 0;
    characteristic.setValue(new byte[]{(byte) value});
    writeCharacteristic(characteristic);
}
```

Se o utilizador estiver nas secções de bateria ou sistema de tração, é possível observar a mudança dos valores. Foi utilizada a API das SharedPreferences, classe usada para fazer cache dos valores medidos, para subscrever a alterações de valores:

```
// Subscrever a alterações na cache
prefs.registerOnSharedPreferenceChangeListener(this);
```

Quando os dados da bateria ou do sistema de tração são atualizados localmente, a classe Cache notifica o Presenter que por sua vez notifica o BatteryFragment ou EngineFragment com os novos dados:

```
// Método invocado quando ocorre uma atualização do valor
@Override
public void onSharedPreferenceChanged(SharedPreferences sharedPreferences,
    String key) {
    if (listener == null) {
        return;
    }
    if (key.equals("engine")) {
        listener.onEngineDataChanged(getEngineData());
    }
    if (key.equals("battery")) {
        listener.onBatteryDataChanged(getBatteryData());
    }
}
```

A aplicação também atualiza a sua localização com recurso aos serviços do Google Play. A localização só é atualizada enquanto o serviço de Bluetooth estiver a executar e houver ligações aos sensores. Como o utilizador se pode deslocar sem estar no carro, não faria sentido o serviço de localização continuar ativo, porque não iria refletir a posição do carro, mas sim do utilizador.

A localização é atualizada a cada 10 segundos enquanto o telemóvel estiver ligado a pelo menos um dos sensores. Caso o utilizador saia do veículo e haja outro telemóvel a recolher dados, este pode ver a localização a ser atualizada e acompanhar no mapa. A atualização da localização é feita com a classe LocationUpdater, que é passada para a BleThread. Quando o serviço é parado, a classe deixa de pedir atualizações.

3.4.6 Notificações

As notificações implementadas nesta arquitetura dizem respeito ao nível da carga da bateria. Quando a carga foi inferior a 10% ou igual a 100%, as funções do Firebase enviam uma notificação para todos os *tokens* registados na base de dados. O registo do *token* é feito no MainPresenter, quando o utilizador abre a aplicação:

```
String token = FirebaseInstanceId.getInstance().getToken();
if (token != null) {
    profileRepository.saveNotificationToken(token);
}
```

Para gravar o *token*, guarda-se uma chave com valor vazio.

```
public void saveNotificationToken(String token) {
  databaseReference.child("notifications")
                .child("tokens")
                .child(token)
                .setValue("");
}
```

Na Figura 3.22 podemos ver o ecrã do perfil do utilizador, onde é possível desativar ou ativar as notificações pretendidas e terminar sessão da aplicação.

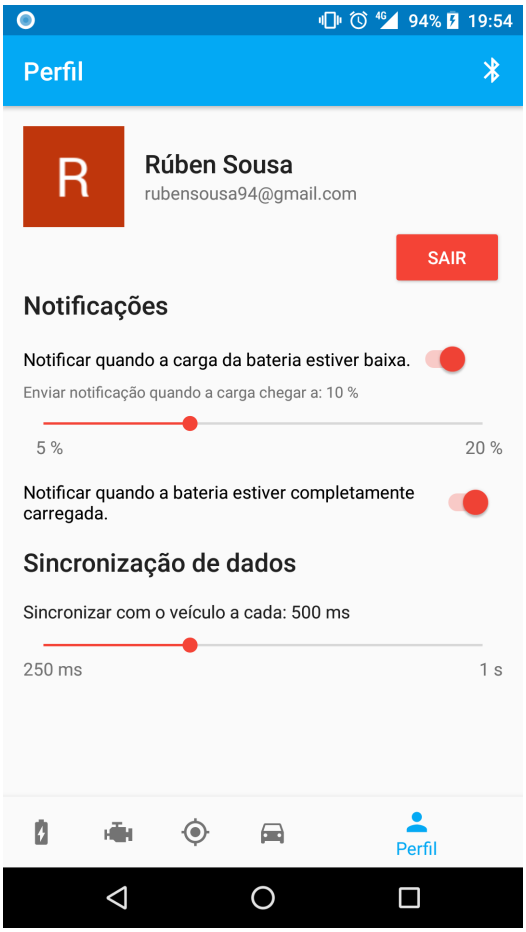


Figura 3.22: Ecrã do perfil do utilizador.

Se o utilizador terminar sessão através do botão, a cache armazenada é invalidada e é redirecionado para o ecrã de autenticação. Quando o utilizador clica num dos Switches das notificações, é feita uma escrita para o Firebase com a mudança do valor:

```
public void enableLowNotification(boolean enable) {
    databaseReference.child("notifications")
            .child("lowNotification").setValue(enable);
```

```
}
public void enableMaxNotification(boolean enable) {
    databaseReference.child("notifications")
            .child("maxNotification").setValue(enable);
}
```

Quando ocorre uma escrita de 100 no valor de carga da bateria, a função do Firebase só envia a notificação se o utilizador a ativou neste ecrã. Na Figura 3.23 podemos ver a notificação recebida.

Figura 3.23: Notificação de carga completa.

Capítulo 4

Testes e Resultados

Neste capítulo, são apresentados os resultados da recolha de dados, com recurso a imagens da aplicação e testes relevantes que demonstram o desempenho do sistema desenvolvido.

4.1 Recolha de Dados

Tal como referido na secção 3.3, a recolha de dados é feita a partir de sensores emulados com um programa em Java, executado num computador à parte, como pode ser visto na Figura 4.1.

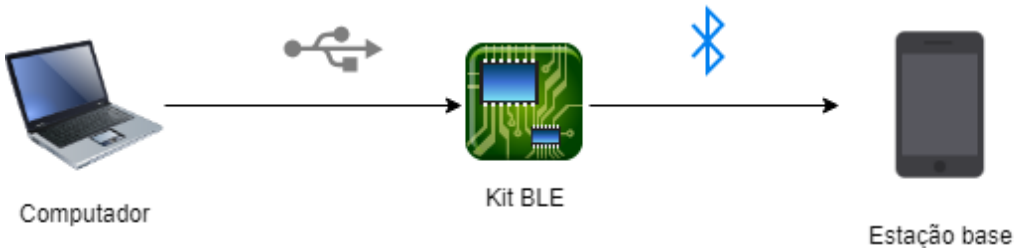


Figura 4.1: Cenário de testes de recolha de dados.

A emulação dos dados é feita porque na altura da escrita os sistemas

sensores não tinham o protocolo de comunicação descrito na secção 3.2.2 implementado. Os dados são recebidos pela porta série (UART) do *kit* BLE e de seguida enviados para a aplicação via BLE. Os sensores emulados são os seguintes:

- Sistema de bateria.
- Sistema de tração.
- Sistema de carregamento.

A localização que é obtida através dos Serviços do Google Play é real.

Na Figura 4.2 está representado o ecrã de informações do sistema da bateria depois de terem sido recolhidos os dados.

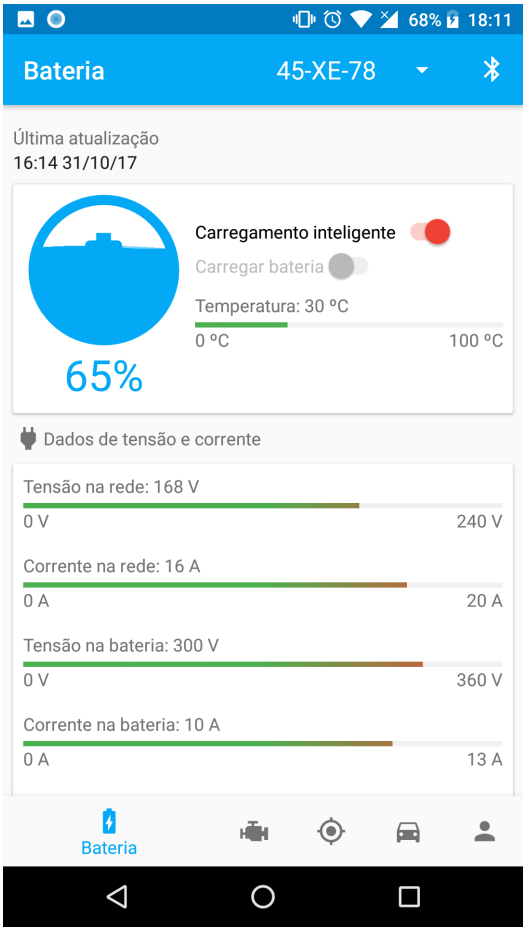


Figura 4.2: Ecrã de informações da bateria.

Neste ecrã é possível consultar todos os dados recolhidos via BLE do sistema da bateria e alterar o estado de carregamento da bateria. A opção de carregamento inteligente faz com que a corrente indicada pelo sistema residencial seja usada para o cálculo da corrente disponível para o carregamento da bateria. Caso essa opção esteja desativada, é possível definir manualmente o estado de carregamento com a opção abaixo "Carregar bateria".

Na Figura 4.3 estão indicados os dados recolhidos pelo sistema de tração.

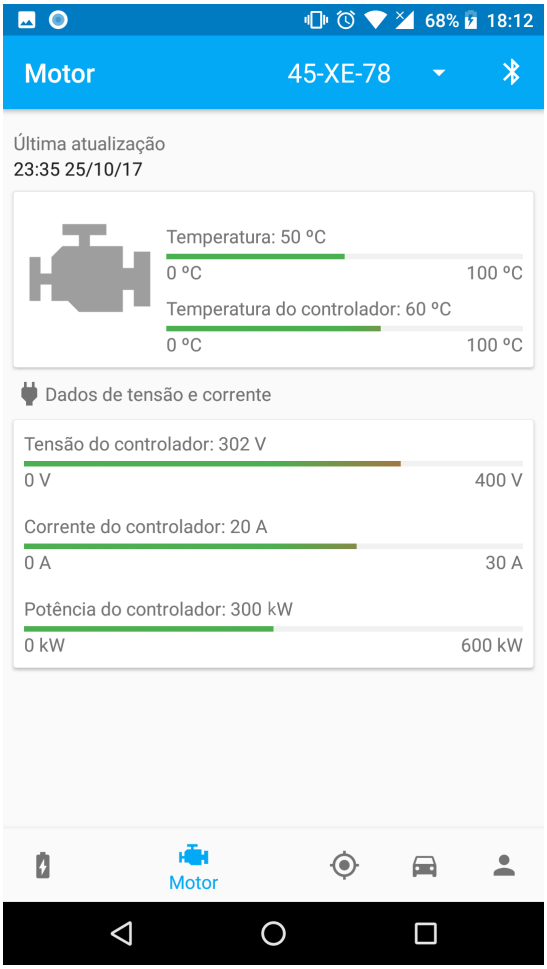


Figura 4.3: Ecrã de informações do motor.

Na Figura 4.4 é possível ver o mapa do Google Maps com a localização de um veículo.

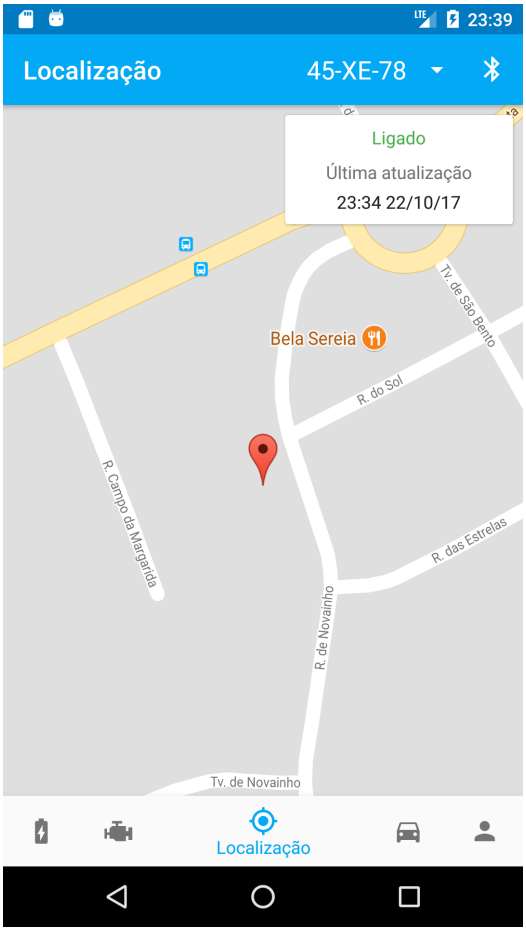


Figura 4.4: Ecrã de localização do veículo.

4.2 Medição de Atrasos na Comunicação

Foi medido o atraso entre a escrita de um valor no Firebase e a receção num sistema sensor do veículo emulado. Este atraso tem de ser baixo para que o sistema reaja rapidamente a alterações no valor de corrente disponível para o carregamento, de forma a que não dispare o disjuntor. O cenário de teste pode ser visto na Figura 4.5.

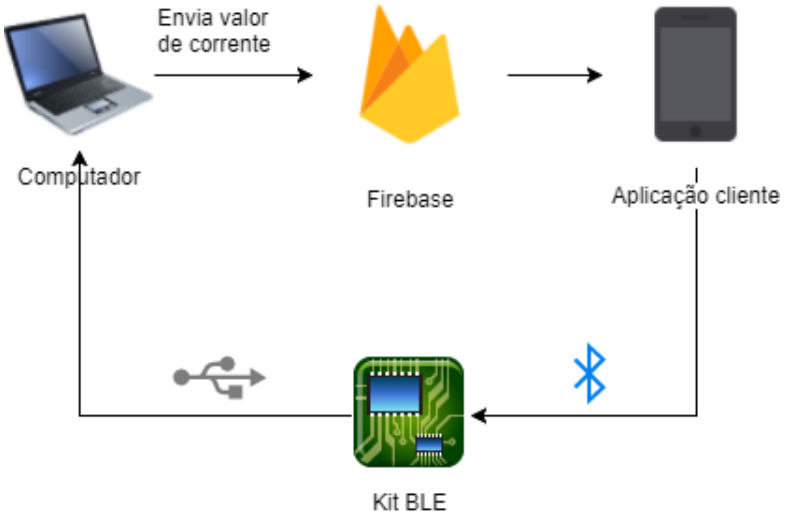


Figura 4.5: Cenário de teste para o cálculo do atraso entre o Firebase e o sensor emulado.

Neste cenário, foram feitos testes com a aplicação cliente ligada via Wi-Fi e dados móveis. O computador gerou 1000 valores de corrente, que por sua vez causaram 1000 receções no sensor emulado. Os testes foram feitos num *smartphone* Nexus 5X perto de um *router* Wi-Fi e com suporte a 4G. O computador utilizado tinha uma ligação por cabo à Internet de 100 Mbps de *upload* e *download* e não tinha qualquer programa a usar exaustivamente a ligação à Internet. Na Tabela 4.1 podem ser vistos os resultados dos atrasos calculados (em milissegundos) com o telemóvel ligado via Wi-Fi e dados móveis.

Tabela 4.1: Atrasos calculados com os dois tipos de ligação à Internet.

Ligação à Internet	Mínimo	Médio	Máximo	Desvio padrão
Wi-Fi	178	257,6	1081	56,31
4G	179	230,4	991	50,97

Foram obtidos melhores resultados quando a aplicação é usada com uma ligação à Internet via 4G. As Figuras 4.7 e 4.6 contêm histogramas que ilustram de melhor forma os resultados obtidos para as 1000 amostras.

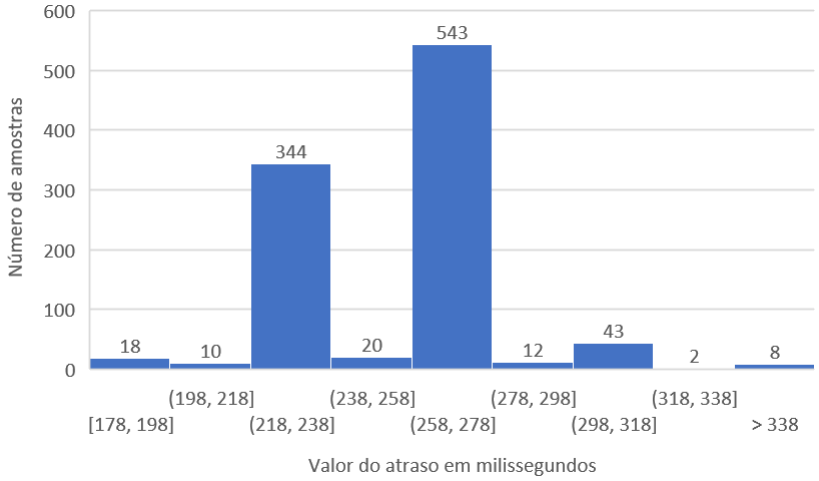


Figura 4.6: Histograma dos valores de atraso com ligação à Internet via Wi-Fi.

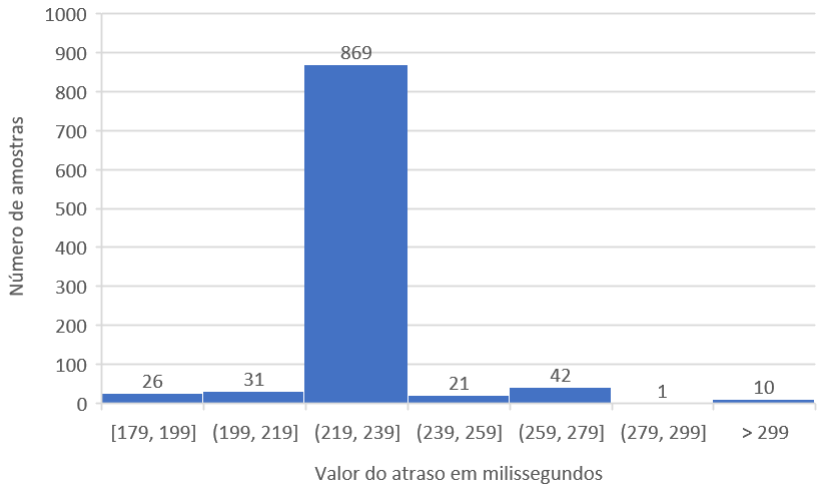


Figura 4.7: Histograma dos valores de atraso com ligação à Internet via 4G.

Com a ligação à Internet via 4G, cerca de 86% das mensagens demoraram menos de 240 milissegundos a serem entregues ao sistema de carregamento e menos de 1% das mensagens demoraram mais de 300 milissegundos. Em ambos os casos, quer via Wi-Fi ou dados móveis, menos de 1% das mensagens demoraram menos de meio segundo a serem entregues, o que permite ao sistema ajustar rapidamente a corrente disponível para carregamento sem que o disjuntor na habitação dispare.

4.3 Medição do Consumo de Dados

Para medir o consumo de dados, foi usada a ferramenta "Android Device Monitor" que está incluída no Android Studio. Na versão 3.0 do Android Studio, foi introduzida uma funcionalidade chamada "Network Profiler", mas à data de realização desta dissertação não existia suporte para tráfego de rede sem ser pelo protocolo HTTP. Visto que o SDK do Firebase usa WebSockets, o tráfego não era detetado nessa ferramenta. O tráfego BLE não foi contabilizado porque não tem custos associados.

Foram feitos dois testes, ambos com duração de 1 minuto. No primeiro teste foi estabelecida ligação à estação periférica do sistema da bateria e a aplicação recebia constantemente notificações de alteração do valor de corrente disponível para carregamento. Estas notificações são enviadas sempre que o valor de corrente é alterado pelo programa do Raspberry Pi que está ligado ao sensor de corrente residencial. O consumo de dados via Internet deste teste pode ser visto na Figura 4.8.

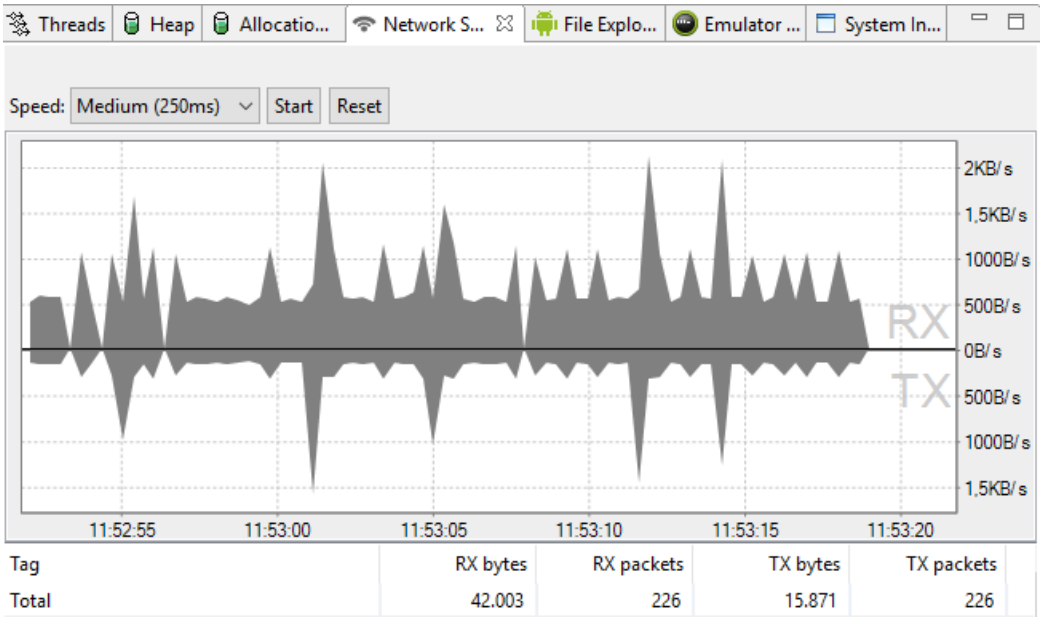


Figura 4.8: Consumo de dados com subscrições de valor de corrente.

O tráfego total nesse minuto foi de 57,874 KB. Por mês, dassumindo o pior caso, e que o sistema estivesse sempre a funcionar, isto daria um consumo total aproximado de: $57,874 \times 60 \times 24 \times 30 \approx 2,38$ GB por veículo. Este tráfego inclui também as atualizações de localização.

No segundo teste, foi apenas desativada a notificação de novo valor de corrente, restando apenas as mensagens do sistema da bateria (Figura 4.9).

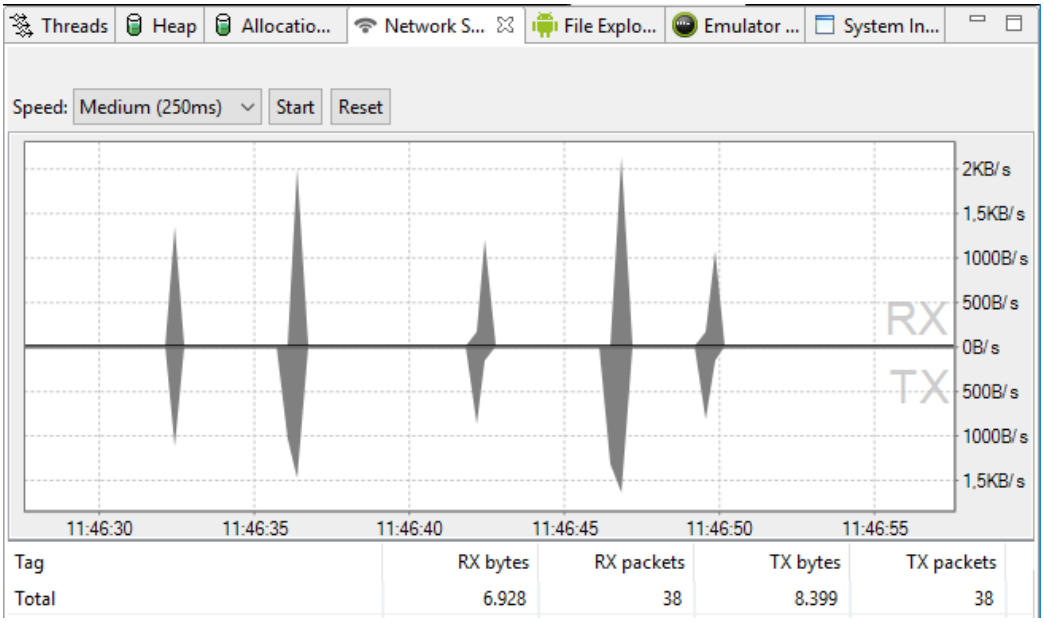


Figura 4.9: Consumo de dados sem subscrições de valor de corrente.

Neste caso, o tráfego total foi de 15,327 KB, o que representa uma redução de aproximadamente 74% em comparação com o teste anterior. Por mês, neste caso, o consumo total aproximado seria de: $15,327 \times 60 \times 24 \times 30 \approx 0,63$ GB por veículo.

Em ambos os cenários, o tráfego total do veículo permite usar o plano gratuito do Firebase, visto que não atinge os 20 GB de transferência de dados. Para 10.000 veículos a enviarem 2,33 GB por mês, teríamos um preço total de: $(10.000 - 20) \times 2,38 = 23.752,4$€. Caso esses 10.000 veículos enviassem 0.62 GB por mês: $(10.000 - 20) \times 0,63 = 6.287,4$€.

Uma das formas de reduzir este volume de tráfego seria guardar apenas em tempo real os dados que importam ao utilizador, tais como o estado de carregamento da bateria e a percentagem da carga. Para os outros dados, tais como a tensão e corrente do controlador do sistema de tração, podia-se apenas guardar localmente no telemóvel e não no Firebase.

Capítulo 5

Conclusão e Trabalho Futuro

Neste último capítulo são apresentadas as conclusões da dissertação, um resumo do trabalho desenvolvido, os problemas encontrados e sugestões de trabalho futuro.

5.1 Conclusão

A arquitetura desenvolvida para esta dissertação cumpriu com todos os objetivos definidos. O principal objetivo era a sincronização dos dados do veículo para a Internet para que estes pudessem ser consultados remotamente. Foram apresentados diversos serviços *cloud* para a integração com a aplicação móvel, dos quais foi escolhido o Firebase pela facilidade de integração, caraterísticas e preço, visto que inclui um plano gratuito sem necessidade de associar um cartão de crédito.

Para a obtenção dos dados do veículo, foram configuradas duas estações periféricas, uma para o sistema de bateria e outra para o sistema de tração. Estas estações periféricas recolhem os dados via sensores emulados com um programa Java através de uma ligação UART. A taxa de refrescamento de

dados é configurável a partir da aplicação. No entanto, ocasionalmente, o *kit* BLE deixava de anunciar o serviço BLE e a aplicação não o reconhecia. Este problema é mencionado no trabalho futuro.

O Firebase foi configurado de forma a restringir o acesso de dados do veículo ao utilizador que o adicionou. Desta forma, cada utilizador só consegue alterar ou ler os dados do seu próprio veículo. O método de autenticação escolhido foi o da Google, visto que todos os utilizadores Android necessitam de uma conta da Google para instalarem aplicações a partir da loja oficial.

Foi desenvolvido um sistema de carregamento inteligente para o veículo, com recurso a um sensor de corrente emulado na habitação. Foi configurada outra estação periférica que obtém dados desse sensor emulado e que se liga via BLE a um Raspberry Pi, que por sua vez está ligado à Internet. O Raspberry Pi executa um programa que sincroniza em tempo real a corrente disponível para carregamento. Após alguns testes, foi medido um atraso médio de 230 ms desde a receção no Raspberry Pi à receção no sensor de carregamento, com o telemóvel ligado via 4G à Internet. Este atraso permite ao sistema reagir a tempo de forma a evitar disparar o disjuntor da habitação.

Para a visualização dos dados do veículo, foi desenvolvida uma aplicação móvel em Android. A aplicação sincroniza os dados que obtém a partir das estações periféricas e mostra-os ao utilizador. Ao mesmo tempo, os dados são enviados para o Firebase e para uma *cache* local, de forma a que possam ser consultados sem ligação à Internet. Para evitar um consumo excessivo de dados, a sincronização com o Firebase é feita a uma frequência menor do que a frequência de receção de dados via BLE. A aplicação também permite ao utilizador configurar notificações do estado de carga da bateria, para que este saiba se o seu carro tem a bateria completamente carregada ou se o precisa de carregar.

Este trabalho deu origem a um artigo [38] publicado no International Workshop on IoT applications in Intelligent Transportation Systems and Logistics, no âmbito da conferência IEEE SOLI 2017.

5.2 Trabalho Futuro

Os veículos elétricos tomarão conta das estradas nos próximos anos e, num futuro próximo, andarão de forma autónoma na estrada. Deixaremos de ser condutores e passaremos a ser passageiros do nosso próprio carro. Assim, torna-se importante desenvolver soluções que usem todo o potencial da tecnologia para o bem do utilizador.

No caso desta dissertação, as sugestões mais relevantes a tratar no futuro, na minha opinião, são:

- Autenticar todas as estações periféricas. De momento, qualquer pessoa com a aplicação móvel consegue-se ligar ao carro. É necessário haver algum tipo de autenticação para que um atacante não possa comutar algum estado do veículo.

- Garantir que as estações periféricas estão sempre a anunciar o serviço BLE quando nenhum dispositivo se encontrado ligado. De vez em quando, a aplicação não reconhecia as estações periféricas.

- Sincronizar apenas os dados relevantes para o utilizador. Dados como a tensão e corrente da bateria podem não ser tão relevantes para o utilizador. Ao sincronizar apenas os dados como a carga da bateria, é possível poupar bastante no consumo de dados (quer dados móveis como tráfego contabilizado pelo Firebase).

- Medir o atraso da receção de dados entre a estação periférica BLE e o Raspberry Pi. O atraso medido no capítulo de testes não inclui o tempo que demora a transmissão BLE entre o Raspberry Pi e a estação periférica, apesar de este tempo ser significativamente menor.

- Medir o consumo de dados com todas as estações periféricas implementadas. Só foi medido o consumo do serviço de localização e da sincronização dos dados de bateria.

- Fazer testes com os sensores reais do CEPIUM.

- Verificar se o carro se encontra trancado ou não e permitir a abertura e fecho remotamente. Neste caso, o telemóvel funcionaria como uma chave do carro.

Para temas futuros que envolvam veículos elétricos autonómos, seria interessante poder usar o telemóvel para chamar o veículo para um dado local. Um exemplo que permitiria ao condutor poupar tempo seria o de evitar a procura de estacionamento. Assim que o condutor chegasse ao destino, podia sair do carro e este estacionar-se-ia sozinho. Quando acabasse de estacionar, enviaria uma notificação para o telemóvel com o local de estacionamento. Assim que o condutor quisesse voltar a usar o carro, voltaria a usar a aplicação para o chamar para a sua localização atual.

Referências

[1] Brahima Sanou. *ICT Facts and Figures 2016*. URL: http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2016.pdf (acedido em 05/12/2016).

[2] Dave Evans. *The Internet of Things - How the Next Evolution of the Internet Is Changing Everything*. White Paper. URL: http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf (acedido em 05/12/2016).

[3] Rita Baldaia da Costa e Silva. *Interface Homem-Máquina para Carro Elétrico baseada em Bluetooth Low Energy*. Dissertação de Mestrado, Mestrado Integrado em Engenharia de Telecomunicações e Informática, Universidade do Minho. 2016.

[4] Rita B. C. Silva, José A. Afonso e João L. Afonso. *Development and Test of an Intra-Vehicular Network based on Bluetooth Low Energy*. Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering 2017, pp. 508-512, Londres, Reino Unido. 5-7 de Julho 2017.

[5] D. Pedrosa, V. Monteiro, H. Gonçalves, J. S. Martins e J. L. Afonso. *A Case Study on the Conversion of an Internal Combustion Engine Vehi-*

cle into an Electric Vehicle. IEEE VPPC Vehicle Power and Propulsion Conference, pp.1-5. Outubro 2014.

[6] José A. Afonso, António F. Maio e Ricardo Simões. *Performance Evaluation of Bluetooth Low Energy for High Data Rate Body Area Networks*. Wireless Personal Communications, Vol. 90, No. 1, pp. 121-141. Setembro 2016.

[7] *Core Bluetooth Overview*. URL: `https://developer.apple.com/library/content/documentation/NetworkingInternetWeb/Conceptual/CoreBluetooth_concepts/CoreBluetoothOverview/CoreBluetoothOverview.html` (acedido em 11/03/2017).

[8] *Smartphone OS Market Share, 2017 Q1*. URL: `https://www.idc.com/promo/smartphone-market-share/os` (acedido em 05/12/2016).

[9] *Android Studio - The Official IDE for Android*. URL: `https://developer.android.com/studio/index.html` (acedido em 05/12/2016).

[10] *Activity*. URL: `https://developer.android.com/reference/android/app/Activity.html` (acedido em 25/03/2017).

[11] *Fragment*. URL: `https://developer.android.com/reference/android/app/fragment.html` (acedido em 25/03/2017).

[12] *Services*. URL: `https://developer.android.com/guide/components/services.html` (acedido em 25/03/2017).

[13] *Descrição geral do serviço Hub IoT do Azure*. URL: `https://docs.microsoft.com/pt-pt/azure/iot-hub/iot-hub-what-is-iot-hub` (acedido em 10/10/2017).

[14] *Microsoft Azure IoT SDKs*. URL: `https://github.com/Azure/azure-iot-sdks` (acedido em 10/10/2017).

[15] *Preços de Hub IoT*. URL: https://azure.microsoft.com/pt-pt/pricing/details/iot-hub/ (acedido em 10/10/2017).

[16] *Preços de Funções*. URL: https://azure.microsoft.com/pt-pt/pricing/details/functions/ (acedido em 10/10/2017).

[17] *Preços de Azure Cosmos DB*. URL: https://azure.microsoft.com/pt-pt/pricing/details/cosmos-db/ (acedido em 10/10/2017).

[18] *Notification Hubs Overview*. URL: https://msdn.microsoft.com/library/jj927170.aspx (acedido em 10/10/2017).

[19] *AWS IoT*. URL: https://aws.amazon.com/pt/iot-platform/ (acedido em 11/10/2017).

[20] *AWS Lambda*. URL: https://aws.amazon.com/pt/lambda/ (acedido em 11/10/2017).

[21] *Detalhes de preço do Lambda*. URL: https://aws.amazon.com/pt/lambda/pricing/ (acedido em 11/10/2017).

[22] *Definição de preço do Amazon DynamoDB*. URL: https://aws.amazon.com/pt/dynamodb/pricing/ (acedido em 11/10/2017).

[23] *Conceitos básicos do Amazon SNS*. URL: https://aws.amazon.com/pt/sns/getting-started/ (acedido em 10/10/2017).

[24] Zhigang Liu, Anqi Zhang e Shaojun Li. *Vehicle Anti-theft Tracking System Based on Internet of Things*. IEEE International Conference on Vehicular Electronics and Safety (ICVES), Dongguan, China. Julho 2013.

[25] V. Monteiro, J. P. Carmo, J. G. Pinto e J. L. Afonso. *A Flexible Infrastructure for Dynamic Power Control of Electric Vehicle Battery*. IEEE Transactions on Vehicular Technology, vol.65, no.6, pp.4535-4547. 2016.

[26] Libelium. *50 IoT real case studies - Smart Parking California*. White Paper, Página 48. URL: http://www.libelium.com/libelium-presents-a-white-paper-with-50-real-iot-success-stories-after-ten-years-of-experience-in-the-market/.

[27] E. Husni, G. B. Hertantyo, D. W. Wicaksono, F. C. Hasibuan, A. U. Rahayu e M. A. Triawan. *Applied Internet of Things (IoT): Car monitoring System Using IBM BlueMix*. International Seminar on Intelligent Technology and Its Application, Lombok, Indonésia. Julho 2016.

[28] Libelium. *Reading Beehives: Smart Sensor Technology Monitors Bee Health and Global Pollination*. URL: http://www.libelium.com/temperature-humidity-and-gases-monitoring-in-beehives (acedido em 12/10/2017).

[29] Gautier Mechling. *IoT - Home automation with Android Things and the Google Assistant*. URL: http://nilhcem.com/android-things/google-assistant-smart-home (acedido em 12/04/2017).

[30] *Cloud Functions for Firebase Sample Library*. URL: https://github.com/firebase/functions-samples (acedido em 20/07/2017).

[31] *PSoC 4 Bluetooth® Low Energy (BLE) 4.1 Compliant Pioneer Kit*. URL: http://www.cypress.com/documentation/development-kitsboards/cy8ckit-042-ble-bluetooth-low-energy-ble-pioneer-kit (acedido em 10/10/2017).

[32] *PSoC® Creator™ Integrated Design Environment (IDE)*. URL: http://www.cypress.com/products/psoc-creator-integrated-design-environment-ide (acedido em 10/10/2017).

[33] *PSoC Timer Example*. URL: https://eewiki.net/display/microcontroller/PSoC+Timer+Example (acedido em 11/10/2017).

[34] *Official jSSC (Java Simple Serial Connector) repository.* URL: https://github.com/scream3r/java-simple-serial-connector (acedido em 15/10/2017).

[35] *TinyB - The Tiny Bluetooth LE library.* URL: http://iotdk.intel.com/docs/master/tinyb/java/ (acedido em 25/01/2017).

[36] *Add the Firebase Admin SDK to Your Server.* URL: https://firebase.google.com/docs/admin/setup (acedido em 25/01/2017).

[37] Michael Haugk. *Starting with Bluetooth LE on the Raspberry Pi.* URL: http://fam-haugk.de/starting-with-bluetooth-le-on-the-raspberry-pi (acedido em 25/01/2017).

[38] J. A. Afonso, R. A. Sousa, J. C. Ferreira, V. Monteiro, D. Pedrosa e J. L. Afonso. *IoT System for Anytime/Anywhere Monitoring and Control of Vehicles' Parameter.* International Workshop on IoT applications in Intelligent Transportation Systems and Logistics. Setembro 2017.